Accelerating Radio Astronomy with Auto-Tuning

Sclocco, A.

2017

document version
Publisher's PDF, also known as Version of record

Accelerating Radio Astronomy with Auto-Tuning

Alessio Sclocco

Cover by Alessio D'Arielli

VRIJE UNIVERSITEIT

Accelerating Radio Astronomy with Auto-Tuning

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad Doctor aan
de Vrije Universiteit Amsterdam,
op gezag van de rector magnificus
prof.dr. V. Subramaniam,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de Faculteit der Bètawetenschappen
op woensdag 11 oktober 2017 om 11.45 uur
in de aula van de universiteit,
De Boelelaan 1105

door

Alessio Sclocco

geboren te Pescara, (Italië)

promotor: prof.dr. H.E. Bal
promotor: prof.dr. R.V. van Nieuwpoort

members of the thesis committee:	prof. dr. Wan Fokkink
	prof. dr. Henk Corporaal
	prof. dr. Aske Plaat
	dr. John Romein
	prof. dr. Ben Stappers

Contents

Chapter 1

Introduction

Nowadays, computers are essential for experimental science, and at the same time scientific experiments are one of the driving forces of technological innovation. In fact, the great scientific experiments of our times, the ones redefining the boundaries of modern science, such as the Large Hadron Collider (LHC) [1], the Square Kilometre Array (SKA) [2], or the advanced Laser Interferometer Gravitational-wave Observatory (LIGO) [3], are not just pushing scientific boundaries farther, but are also pushing computer science and engineering farther. Technology and experimental science have become interdependent. On one side, without all the advancements in computer science and high-performance computing, advancements in areas like hardware design, networking, storage, and algorithms, it would not be possible to process all the data generated by these experiments. On the other side, these experiments will not generate only answers, but new questions too, and answering those questions will require the design of new experiments, with new and more precise instruments, starting the cycle of innovation again.

In this thesis, we focus on a particular scientific domain: radio astronomy. Using radio astronomy, we show the interdependence between this field of experimental science, and our own domain of computer science and high-performance computing: this process of enhancing sciences with computer science is what we nowadays call eScience. We begin by showing how many-core accelerators and auto-tuning can be used to speed up some specific radio astronomy applications, and how these accelerated applications can be used to satisfy the observational constraints of real world telescopes. In this way, we highlight how advancements in computer science can be used for the enhancement of science. We continue by showing that tuning complex applications on many-core accelerators is challenging, but at the same time that tuning these applications is necessary to achieve the performance needed in the real world. Therefore, we also highlight how the

computing challenges of radio astronomy provide research questions for us as computer scientists. Although the reference domain throughout this thesis is radio astronomy, we believe that our results can be generalized and can be applied to many different domains. The rest of this chapter will be used to introduce the research questions that we aim to answer with this thesis, in Section 1.1, and to present the overall structure of the thesis itself, in Section 1.2.

1.1 Research Questions

Throughout this thesis, our goal is to show how auto-tuning and many-cores can be used to accelerate radio astronomy. Auto-tuning is an optimization technique that consists of automatically finding the best values for some performance-relevant parameters. In the context of this thesis these parameters are the configuration knobs of various radio astronomy applications. If there is already a history of using Graphics Processing Units (GPUs) to accelerate scientific workloads, such as in [4], [5], or [6], the use of auto-tuning in scientific applications is less common, thus our goal is to show how many-core accelerators and auto-tuning can be combined for the benefit of experimental radio astronomy. To achieve such a goal in the most rigorous way, we need to define a set of questions that can be used to guide our research into this field. We begin by defining an overall research question in Section 1.1.1. This main research question will be the focus of this thesis, but we will not answer it directly. What we are going to do here is to follow a bottom-up approach, thus inductively build up knowledge that can be used, farther along the road, to address the main question itself. Therefore, from Section 1.1.2 to Section 1.1.5, we define four sub-questions of increasing reach and complexity, building on the answers of the previous ones, and laying the foundations of the following ones. Only after answering all these sub-questions, we will turn our attention back to the main research question and provide an answer to that. Below, we give the list and description of all the research questions, starting with the main one.

1.1.1 How Can Auto-Tuning Accelerate Radio Astronomy?

This research question, the overall research question of this thesis, follows straight from the title and main goal of this work. This goal is to determine how auto-tuning can be used to accelerate radio astronomy, and therefore this is the main question that we need to answer with this thesis. Auto-tuning is a well-known optimization in computer science, and it has been successfully used in various domains, but how important is it for radio astronomy? Is the real-time processing of enormous quantities of data, typical of radio astronomy, going to benefit from auto-tuning, and how much? Is auto-tuning just another optimization technique,

or is it necessary to a point where not using it would make it almost impossible to achieve the scientific goals of our era? In this thesis, we will provide theoretical analysis and empirical data to answer these questions.

1.1.2 RQ1: Can Many-Cores Be Used to Accelerate Radio Astronomy Algorithms?

The first sub-question that we address in this thesis is about using many-cores to accelerate single radio astronomy algorithms. Many-core accelerators have gone from special purpose processors, used mainly for computer graphics, to accelerators for high-performance computing, used in different domains and available in many supercomputers. But are they the right tool for the specific needs of radio astronomy? Although parallelism is a natural component of many radio astronomy algorithms, most of these algorithms are memory-bound and do not really benefit from the high arithmetic throughput provided by these platforms. In order to answer this question, we will implement different radio astronomy algorithms on a variety of platforms, from multi-core systems to many-core accelerators like GPUs and the Intel Xeon Phi, and compare their performance and achieved speedups with more traditional solutions based on CPUs. The radio astronomy algorithms that we will implement to answer this question are: beam forming, dedispersion, folding, and signal-to-noise ratio computation. We did choose these algorithms because of their importance for two instruments operated by the Netherlands Institute for Radio Astronomy (ASTRON): LOFAR and Apertif. Overall, the algorithms that we selected in this thesis represent the majority of the execution time of LOFAR (see [7] and [8]) and the Apertif fast radio bursts pipeline (see [9]).

1.1.3 RQ2: What Is the Impact of Auto-Tuning on Radio Astronomy Algorithms?

Along with determining whether many-core accelerators are a viable solution to accelerate radio astronomy algorithms, we want to understand the impact of auto-tuning on these same algorithms. How important is it to tune these algorithms? What is the performance gain achieved with auto-tuning? Does auto-tuning provide performance portability between different platforms? To answer these questions, we will tune the radio astronomy algorithms listed in Section 1.1.2, and measure the performance improvement, if present, caused by tuning. Tuning will take place for different use-case scenarios and different platforms. We will also measure how important auto-tuning is for these algorithms, to determine if this optimization is necessary or not, and will investigate whether auto-tuning can be used to provide performance portability across different accelerators.

1.1.4 RQ3: Are Many-Core Accelerators and Auto-Tuning Useful for Complex Radio Astronomy Pipelines?

After answering the questions presented in Section 1.1.2 and 1.1.3, and having therefore established if many-core accelerators and auto-tuning can and should be used to accelerate radio astronomy algorithms, we now focus on complex pipelines made of these algorithms. The reason is that in radio astronomy, like in many other scientific fields, a single algorithm is rarely used in isolation, and is used as part of a larger pipeline instead. We aim to investigate if it is possible to implement whole many-core accelerated radio astronomy pipelines, and whether it is possible to tune not just all the components in isolation, but also the pipeline as a whole, as it happens in other domains such as image processing [10]. To answer this question, we will use some of the algorithms, that we already implemented and tuned, and combine them into scientific pipelines. We will measure the performance of these pipelines, and determine if they can satisfy the operational requirements of different real world telescopes.

1.1.5 RQ4: How Difficult Is Auto-Tuning Of Many-Core Accelerators?

The last of the research questions is about auto-tuning, and in particular about how difficult the tuning of many-core accelerators is. In fact, before determining if auto-tuning and many-cores are viable means to accelerate radio astronomy, we need to understand how difficult auto-tuning is. To answer this question, we will analyze the optimization space and the optimal configurations of different algorithms, and measure how difficult finding these optimal configurations is. We will also measure how important it is to find the optimums, and if they are portable between different accelerators and input instances. We use many different algorithms from different domains to show that the answer to this particular question is not limited to the domain of radio astronomy.

1.2 Thesis Outline

In this last section we introduce the content of all the chapters in this thesis, to give the reader a global outline of the thesis, and provide a match between the research questions and the chapters in which they are addressed.

Chapter 2 provides a background on radio astronomy and auto-tuning, two of the main topics of this thesis, and sets the context for all the following chapters. Moreover, Chapter 2 provides a list and describes all the many-core accelerators used throughout the thesis.

The first chapter to address one of the research questions is Chapter 3; this chapter is based on [11]. In this chapter, we study how many-core accelerators can be used to accelerate beam forming, a signal processing algorithm widely used in radio astronomy, radar systems, seismology, acoustics, and wireless networks. Moreover, we show how auto-tuning can be used to improve the performance of a many-core beam former. This chapter addresses, as its main research question, question **RQ1**, but it also provides data to answer questions **RQ2** and **RQ4**. Of the co-authors of [11], Dr. Varbanescu, helped with comments on the paper, and with supervision during the work on [12], while Dr. Mol, at ASTRON, implemented the beam former for the Blue Gene/P, on which the many-core beam former is based.

Chapter 4 introduces a new many-core implementation of the dedispersion algorithm, and presents performance results for this algorithm on various many-core accelerators. Even more important, this chapter provides an analysis of the tuning process of the dedispersion algorithm on many-core accelerators, making it possible to study the impact that auto-tuning has on the algorithm's performance. This chapter addresses two different research questions, **RQ1** and **RQ2**, but it also provides data to answer question **RQ4**. Chapter 4 is based on [13]; of the co-authors of this paper, Dr. Hessels and Dr. van Leeuwen, both astronomers at ASTRON and the Anton Pannekoek Institute (API), provided guidance in the domain specific aspects of dedispersion, and in the definition of the scenarios.

Chapter 5, introduces the reader to a many-core, and auto-tuned, implementation of the Apertif Radio Transient System (ARTS) transients pipeline. This chapter addresses, as its main research question, question **RQ3**. Of the co-authors of [9], on which Chapter 5 is based, Dr. van Leeuwen, being the principal investigator of the time domain experiments on ARTS, provided guidance on the scientific aspects of the pipeline, and on ARTS requirements.

In Chapter 6, based on [14], we introduce a parallel pulsar searching pipeline, and show its performance on two GPUs and a Xeon Phi, using scenarios based on three different real world telescopes. Although similar, this pipeline is more complex and needs more computation than the one presented in Chapter 5. This chapter addresses, as its main research question, question **RQ3**.

Chapter 7, based on a manuscript that is not yet published at the time of this writing, introduces the concepts of tuning difficulty and optimum portability. In this chapter, we look at five different algorithms, some of them memory-bound, some of them bound by the amount of data-reuse that can be exploited during execution, and study the shape of their optimization spaces on various hardware platforms, and for various input sizes. Moreover, we look at how portable the optimal configurations of these algorithms are. This chapter addresses, as its main research question, question **RQ4**.

The final chapter, Chapter 8, presents the conclusions of this thesis, including

the answer to all the research questions introduced in this chapter.

Chapter 2

Background

Although this is a thesis in computer science, it follows a multidisciplinary approach in which computer science and radio astronomy are intertwined, and some knowledge in both fields is necessary to understand all the results and contributions of the following chapters. Therefore, the goal of this chapter is first to introduce the reader to the domain of radio astronomy, and second to explicit the connections between computer science and radio astronomy that are useful for the understanding of this work. Without pretending to write a complete overview of the vast field of radio astronomy, in Section 2.1 we briefly explain what radio astronomy is, and focus in particular on the two radio astronomy problems to the solution of which we contribute with this thesis: the search of pulsars and transient sources.

After providing the reader with the necessary background in radio astronomy, we set the context for auto-tuning, the main optimization technique used throughout this thesis, in Section 2.2. Instead of focusing on all the details of auto-tuning, or providing a review of the field that is outside of the scope of our work, we explain why we believe that auto-tuning is of particular importance for radio astronomy. We believe that, after reading this chapter, the reader with a background in astronomy will better understand why an optimization like auto-tuning should be of interest to him or her, while the reader with a computer science background will have better familiarity with the terminology of radio astronomy, and understand the connection between radio astronomy and auto-tuning.

To conclude the chapter, in Section 2.3 we list and introduce all the multi-core CPUs and many-core accelerators used in the following chapters. While introducing these platforms and their characteristics, we explain the reasons that make us believe that many-cores are suitable platforms to accelerate radio astronomy

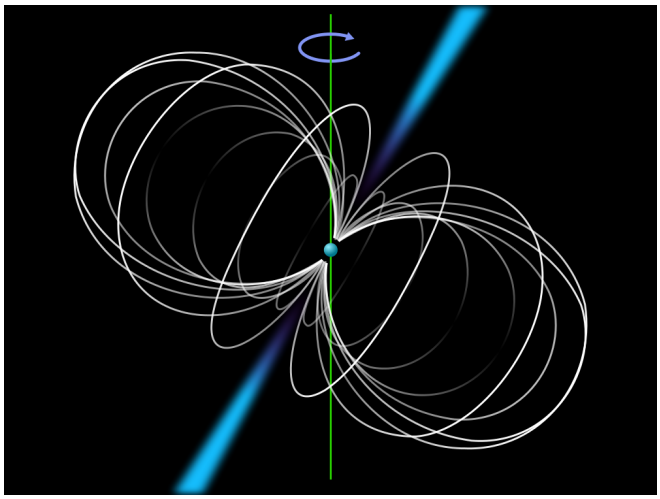

Figure 2.1: Schematic view of a Pulsar, courtesy of Roy Smit on Wikipedia.

algorithms and complex pipelines.

2.1 Radio Astronomy

Radio astronomy is one of the youngest fields of astronomy, the centuries old science that studies the objects that can be observed outside the far reaches of our planet's atmosphere. Radio astronomy was born in the 1930s, when Karl Jansky, investigating some periodic interferences on transatlantic radio transmissions, detected a signal coming from the depths of our galaxy, the Milky Way. Thanks to this discovery, astronomers realized that there is more in the sky than what we can see with our naked eyes, or even the most powerful optical telescopes.

Radio astronomy is, therefore, the field of astronomy that studies the celestial objects, and their associated processes, that are observed in the radio segment of the electromagnetic spectrum. While some of these objects can also be observed in the optical segment of the electromagnetic spectrum, such as stars and galaxies, others are not, and are instead specific to radio astronomy. Among these objects we can list pulsars and fast radio bursts (FRBs).

Pulsars are special highly-magnetized and rapidly-rotating neutron stars; an artistic representation of a pulsar, made by Roy Smit and available on Wikipedia, can be seen in Figure 2.1. The small object at the center of Figure 2.1 is the pulsar, and the white closed lines originating in one of the magnetic poles and ending in the other represent the magnetic field of the pulsar; the rotation axis

of the pulsar is the green line, and the blue emissions originating from the area close to both poles are a representation of the radio signal.

What makes pulsars different from other radio sources is that their emission is not continuous, but rather periodic, and in fact the word "pulsar" is a portmanteau of the words "pulsating" and "star". This periodic emission is caused by having the rotation and magnetic axes not aligned with each other, and so from the point of view of an external observer the two magnetic poles rotate around the rotation axis, and so do the radio emissions associated with them. The result is an effect similar to the one of a lighthouse, with the emission visible only when aligned with our line of sight. From this description, it is clear that the characteristic periodicity of pulsar signals is related to the speed at which pulsars spin.

The first pulsar was discovered in 1967 by Hewish and Bell [15], and this discovery contributed to the awarding of the 1974 Nobel prize in physics to Hewish and Ryle. In this same year, Hulse and Taylor discovered the first binary pulsar, and for this discovery were successively awarded the Nobel prize in physics in 1993 [16]. In binary pulsars, the observed pulsar has a companion, and the interaction between these two objects alters the period of the pulsar itself, an observable effect of gravitation. These, and other properties, are what make pulsars interesting not just for astronomers, but for physicists and other scientists. Because of their precise timing, pulsars can be used to indirectly measure the structure and electron distribution of the inter-stellar medium, better understand the physics of neutron stars, study the interaction of massive objects and relativity, and even detect the passage of gravitational waves.

In the fifty years between the discovery of the first pulsar and the writing of this thesis, only less than three thousand pulsars have been discovered. But to produce the scientific results that we just highlighted, it is necessary to discover, precisely time, and monitor a considerable number of pulsars, and the more exotic, the better. Unfortunately, finding new pulsars is a time consuming process, and it requires state of the art computing facilities as much as it requires modern and highly sensitive radio telescopes. The reason that discovering new pulsars is a time consuming process is that the signal emitted by a pulsar looks very different at the emission point than here on Earth, because the electromagnetic radiation that is emitted by the pulsar gets scattered, dispersed, and in general distorted by the interaction that it has with matter along its path. Although these interferences can be corrected by means of signal processing, the correction process requires parameters like the position of the pulsar, or its distance from Earth, parameters that are not known in advance when looking for new objects. Therefore, the data collected by telescopes need to be blindly processed for every reasonable distance from Earth and position in the sky, and then the resulting processed signals are inspected to detect pulses in them. In Chapters 4 and 6

we discuss the main computational challenges behind discovering new pulsars in more detail, and provide accelerated solutions to some of these challenges.

FRBs are the other objects that we previously mentioned. Like for pulsars, the name of these objects is descriptive of the way in which they are observed, i.e. as milliseconds lasting transient radio emissions. Although we used the word objects to describe FRBs, the correct term for them would be phenomena, as they are most certainly emissions associated with high energetic, and possibly destructive, phenomena, and not celestial objects on their own.

The first FRB was discovered by Lorimer [17] in 2007, in archive data recorded in 2001, and at the time of the writing of this thesis fewer than twenty have been discovered. Their origin is still unknown, and there are currently only theories on what can lay behind these phenomena. Nevertheless, and although they don't have a period that can be used to time them, they are of high importance for astronomers and physicists interested in the high-energetic universe, gravitational waves, and the inter-galactic medium.

Because FRBs are isolated, and possibly not repeating, events, to find new ones it is necessary to constantly survey the sky, and process a sizeable amount of data in real-time. During this thesis, we have been involved in the design and development of the FRB real-time searching pipeline for the Apertif Radio Transient System (ARTS), a pipeline that is described in Chapter 5 of this thesis.

2.2 Auto-Tuning

Tuning, and auto-tuning, are common optimization techniques used in computer science. While a more thorough analysis of auto-tuning will be provided in Chapter 7, in this section we aim to explain why we believe that auto-tuning is important for radio astronomy. In particular, as there are many different forms of auto-tuning [18], here we focus on exhaustive offline auto-tuning. Therefore, in this thesis we define auto-tuning as the technique to automatically find the best configuration of an application's parameters, among all possible valid configurations of the said parameters, before the application is used in production.

Let us assume that we have just parallelized some radio astronomy algorithm, and that at this point we have multiple options regarding which optimizations to apply, and which configuration of the run-time parameters to use to process a certain input while running on a particular, and well-defined, platform. Assuming complete knowledge of the algorithm, the hardware details of the platform, and of the input to process, it would be fairly easy to decide the optimizations to apply. With this knowledge, we could also decide on the specifics of the run-time configuration, such as how many threads to use, and how to partition them. Unfortunately, in the real-world we seldom have all this knowledge in advance.

Lack of knowledge is not the only limiting factor in our quest to a priori find

the best configuration for our algorithm. Another factor is represented by the evolution of scientific requirements and use-case scenarios. The main tool of radio astronomy are radio telescopes, and their operational lives are in the order of years or even decades. Even though the basic operational parameters of a telescope are known during the design phase of the instrument itself, or at least during commissioning, and without considering the real possibility that a telescope can get upgrades during its operational life, the science that is produced during the life of this telescope may not be what was planned for during its construction. In fact, telescopes built years ago to conduct pulsar surveys may now be used to search for FRBs, and could tomorrow be used to identify the objects associated with gravitational waves. Therefore, scientific software needs to easily adapt to new use-cases, and optimizations or run-time configurations cannot be easily determined in advance, especially if they depend, even just partially, on these use-cases.

New use-cases are not the only changes that radio astronomy software has to cope with; another change is represented by the hardware platforms that are used to execute the software, and the rate of change may be even higher. In fact, if telescopes built decades ago are still operational, and if experiments designed years ago are still in progress, the computer systems connected to these telescopes, or simply used to process their data, are certainly not the same used decades ago. Adapting software to new hardware platforms was easier in the past, when the most probable architectural change was just an increase in frequency for CPU or main memory, or an increase in disk capacity or network bandwidth. While many architectural changes in the past were transparent for the user, this is not always the case in our time. Adapting software to new hardware platforms today may require a complete rewrite of the code itself, for example to make it parallel, or to adapt its structure to accommodate more complex memory hierarchies, new vector instructions, or a different organization of the cores. Having to rewrite the code for each new platform is expensive and time consuming. But also a simple refactoring of the code to allow for new optimizations or new execution parameters is time consuming, if the code was not designed from the beginning to facilitate its evolution.

Those are the reasons why we believe that auto-tuning can be important for radio astronomy, and for other scientific domains with long term infrastructures as well. We believe that, by designing software for tuning from the early stages of development, it is possible to adapt more easily to new scientific requirements and to new platforms. In this thesis, we show this approach by parallelizing different radio astronomy algorithms, and even two complete radio astronomy pipelines, without making operational assumptions, and leaving to the final user the choice of which optimizations to use, or not, and how to configure the run-time. We then use auto-tuning to select the best optimizations and run-time configurations for a

variety of platforms, multiple telescopes, and different use-case scenarios. In this way, we show how we can adapt the same software to different situations, mimicking the reality of scientific software used in production environments. Moreover, by optimally tuning our software, we are able to achieve high performance, and because of this we can achieve multiple goals, like satisfying real-time constraints for radio astronomy pipelines, minimizing the size of a cluster, or increasing the precision of the scientific results.

2.3 Many-Core Accelerators

As already mentioned in the previous section, for years many architectural improvements were transparent to the users. The most notable example regards CPUs. With transistors getting smaller every few years, it was possible to add new features into new CPUs, and at the same time increase their operating frequencies. This combination of higher frequencies and new features that are transparent to the user, such as better and larger caches, or improved branch prediction, made software faster without much effort. Obviously, it was also possible to achieve greater improvements in performance by modifying the code, for example using new instructions, or vector operations, but the natural improvement brought by higher frequencies was enough for most scientific workloads.

Nowadays, architectural improvements are not always transparent to the user, and actions are required to translate these improvements into performance gains. One of the most visible architectural changes over the last years has been the introduction to the market of multi-core CPUs. With CPUs operating frequencies decreasing, and core count increasing, it was not anymore possible to exploit architectural improvements without modifying the software to exploit parallelism. At the same time CPU manufacturers began adding more cores to CPUs, Graphics Processing Units (GPUs) manufacturers began making programming these devices more generic and less graphics oriented. In just a decade, multi-core CPUs are the standard, and GPUs are co-processors that may have nothing to do with graphics. Today, the performance capabilities of modern high-performance computing systems are fueled by a combination of multi-core CPUs, most of the times supporting vector instructions, and other accelerators such as GPUs.

In this thesis we are going to address as *many-core accelerators* all parallel processors whose number of cores is, at least, in the order of tens, and to whom all or part of the computation can be offloaded. In the following chapters, we will use different platforms to show how auto-tuning can be used to accelerate radio astronomy, and these platforms will include CPUs, GPUs, and the Intel Xeon Phi. A comprehensive list of all the platforms used throughout the whole thesis is presented in Table 2.1. In this table, together with the name of the platform, we show the year the platform was introduced on the market, its architectural

Platform	Year	Family	Cores	GFLOP/s	GB/s	$\frac{GFLOP/s}{GB/s}$
AMD Opteron 6172	2010	Magny-Cours	12	201	42	4.7
AMD HD6970	2010	TeraScale 3	1536	2793	176	15.8
AMD HD7970	2012	GCN	2048	3788	264	14.3
AMD FirePro W9100	2014	GCN 2	2816	5237	320	16.4
AMD R9 Fury X	2015	GCN 3	4096	8601	512	16.7
Intel Xeon E5620	2010	Westmere	4	76	25	3.0
Intel Xeon E5-2620	2012	Sandy Bridge	6	192	42	4.5
Intel Xeon Phi 5110P	2012	Knights Corner	120	2022	320	6.3
Intel Xeon Phi 31S1P	2013	Knights Corner	112	2006	320	6.2
NVIDIA GTX 580	2010	Fermi	512	1581	192	8.2
NVIDIA GTX 680	2012	Kepler	1536	3090	192	16.0
NVIDIA K20	2012	Kepler	2496	3519	208	16.9
NVIDIA GTX Titan	2013	Kepler	2688	4499	288	15.6
NVIDIA K20X	2013	Kepler	2688	3935	250	15.7
NVIDIA GTX Titan X	2015	Maxwell	3072	6144	336	18.2
NVIDIA GTX 1080	2016	Pascal	2560	8228	320	25.7

Table 2.1: Characteristics of the platforms used throughout the thesis.

family, the number of its cores, the theoretical peak throughput in terms of both single precision floating point arithmetic operations and memory bandwidth, and the ratio between these two throughput metrics. While each chapter will use only a subset of these platforms, and it will reintroduce the important characteristics of each specific platform in the context of that particular chapter, in this section we briefly introduce the differences among them and their general characteristics.

We decided to also include three CPUs among our platforms, one from AMD and two from Intel. Although they are not many-core accelerators, we use them as either baselines for performance, or to show that auto-tuning can make well-structured and parallel software portable from CPUs to many-core accelerators without human intervention. Moreover, when multiple multi-core CPUs are present in the same node, have access to the same memory, and take part in the same parallel computation, they act as many-core accelerators, and are thus interesting to our research.

The GPUs we use in this thesis are from the two main vendors, AMD and NVIDIA, and included are both consumer and professional grade ones. The consumer GPUs are the AMD HD6970, HD7970, and R9 Fury X, together with the NVIDIA GTX 580, 680, Titan, Titan X, and 1080; the professional grade GPUs are the AMD FirePro W9100, and the NVIDIA K20 and K20X. For both vendors we include GPUs from different generations, with the oldest being from 2010 and the newest from 2016. The reason for having GPUs from multiple

vendors, and for each vendor having GPUs from multiple generations, derives from the need for adaptability that we highlighted as essential for radio astronomy in Section 2.2. Having all these different platforms, we can show how it is possible to use auto-tuning to adapt code to GPUs produced by different manufacturers, and to different families of products from a same manufacturer. In fact, in the following chapters we demonstrate how it is possible to achieve high performance for different algorithms, from radio astronomy to other domains, without the need to manually optimize or change the code for any particular platform. Another reason for using different GPUs in this work is that we can compare them to each other, analyze their evolution in the last seven years, and find the best platform for each application.

The architectural families of the GPUs listed in Table 2.1 are the following. The AMD HD6970 is a member of the TeraScale 3 family, the HD7970 a member of the first version of the Graphics Core Next (GCN) family, the W9100 a member of the second version of the GCN family, and the Fury X a member of the third generation of the GCN family. As for NVIDIA, the GTX 580 is a member of the Fermi family, the GTX 680, GTX Titan, K20 and K20X are all members of the Kepler family, the GTX Titan X a member of the Maxwell family, and the GTX 1080 a member of the Pascal family.

The other many-core accelerator included in this thesis is the Intel Xeon Phi. For the Phi we only have two platforms, and because they are both members of the same architectural family, Knights Corner, their characteristics are fairly similar to each other. Although for this accelerator we cannot compare different families to each other, or show how auto-tuning can be used to adapt code between different families, it is still interesting for two main reasons: first we want to understand how this many-core accelerator can be used in the domain of radio astronomy, and second we can test the adaptability provided by auto-tuning between GPUs and the Phi.

After introducing all the platforms we use in this thesis, and having highlighted their importance in the context of our research, we want to briefly focus on a characteristic of these accelerators that will be extremely important in the following chapters: the ratio between their peak arithmetic throughput and memory bandwidth. This value is not just important as a representation of the gap between peak arithmetic throughput and memory bandwidth, that seems to be growing for each new accelerator, but mostly because it can be interpreted as the minimum Arithmetic Intensity (AI) that an algorithm needs to have to not be memory-bound on that platform [19]. The higher the value, the more performance of applications running on this platform will be limited by memory bandwidth. For the three CPUs in our list, this value is between 3 and 5, and so by performing 3 to 5 arithmetic operations per byte accessed in memory applications running on these CPUs are not limited by memory bandwidth anymore.

This ratio is also relatively low, and comparable to the value of the CPUs, for both Xeon Phis. Unfortunately, the memory bandwidth that can be easily achieved in practice on the Xeon Phi is much lower than the theoretical peak [20], and therefore the minimum AI necessary to not be memory-bound is larger.

On average, GPUs have a much larger gap between arithmetic throughput and memory bandwidth than CPUs or the Xeon Phi, and therefore the ratio is also larger. This ratio, for the four AMD GPUs, is between 14 and 17, and year after year, from one architectural family to the next, it is somewhat stable. This means that each new successive architecture sees an increase in both arithmetic throughput and memory bandwidth. The ratio is far less stable for the seven NVIDIA GPUs we use. The Fermi GPU has a ratio between arithmetic throughput and memory bandwidth of just 8, just slightly more than the value of 6 of the Phis, while the Pascal GPU has a ratio of 25, the highest among all the platforms we use in this thesis. The other five GPUs have a ratio between 15 and 19, higher than the AMD GPUs, but comparable. In general, we see that all GPUs, with the exception of the old GTX 580, require tens of arithmetic operations for each byte accessed in memory for an application to not be memory-bound. This high value means that most applications running on GPUs will be memory-bound, even if they were limited by arithmetic throughput on other architectures. Therefore, to achieve high performance on these accelerators, the main focus must be in trying to optimize memory use, and not computation. In the following chapters we will show how, by using auto-tuning and exploiting data-reuse, it is possible to increase the AI of some algorithms, and therefore improve their performance.

Chapter 3

Beam Forming*

Radio telescopes are used to capture the frequencies of the electromagnetic spectrum that are outside the realm of visible light. To answer astronomy's big open questions, for instance those related to the origin of the universe, there is a need for more powerful and precise instruments. Due to engineering limitations and economical constraints, simply building larger telescopes to increase resolution is not viable anymore. An alternative is radio interferometry, a technique that combines signals of multiple receivers, thus creating a single large virtual telescope. To cope with the complexity of this scenario, telescopes rapidly evolve into *software telescopes*, while they used hardware-based processing in the past. The software solution increases flexibility and lowers construction costs. The computational demands are challenging: LOFAR requires tens to hundreds of teraflops. The SKA, a next-generation instrument that is still under construction, will require exaflops. Therefore, high performance and power efficiency are of key importance.

LOFAR is an example of such a software telescope: it has more than 80,000 omni-directional antennas, see Figure 3.1, which are geographically distributed in five different countries. LOFAR's signal processing pipeline is implemented in software, and runs on an IBM Blue Gene/P (BG/P) supercomputer. LOFAR is the largest and most complex radio telescope in the world. It is driven by the astronomical community, which needs a new instrument to study an extensive number of new science cases. Five key science projects have been defined. First, we expect to see the *Epoch of Reionization* (EoR), the time when the first star galaxies and quasars were formed. Second, LOFAR offers a unique possibility in particle astrophysics for studying the origin of high-energy *cosmic rays*. Nei-

*This chapter has been adapted from "Radio Astronomy Beam Forming on Many-Core Architectures" [11]

Figure 3.1: A field with LOFAR antennas.

ther the source, nor the physical process that accelerates such particles is known. Third, LOFAR's ability to continuously monitor a large fraction of the sky makes it uniquely suited to find new *pulsars* and to study *transient sources*. Fourth, *Deep Extra-galactic Surveys* will be carried out to find the most distant radio galaxies and study star-forming galaxies. Fifth, LOFAR will be capable of observing the so far unexplored radio waves emitted by *cosmic magnetic fields*. For a more extensive description of the astronomical aspects of the LOFAR system, see [21].

One of LOFAR's key features is the capability to point at multiple directions in the sky at the same time, without the need to move any mechanical part; this is made possible by its software *beam former*. In fact, beam forming is *the only way* to point the LOFAR telescope. With the high number of antennas, and the ever increasing number of observations requested by the astronomers for their experiments, it becomes apparent that the beam former is a fundamental part of the telescope's pipeline, with particularly demanding requirements of high performance and scalability.

LOFAR's production beam former, at the time of [11], was executed on a Blue Gene/P, together with many other components of the LOFAR pipeline. The beam former is an inherently parallel application. This chapter investigates whether or not it is possible and effective to parallelize the beam former using modern many-core architectures, such as *Graphics Processing Units* (GPUs), and large multi-core CPU nodes. Our aim is to achieve high performance and power

efficiency.

We are interested in GPUs as a possible way to accelerate radio astronomy because in the last ten years they have evolved from simple graphics processors to general purpose computational units, offering a mix of low costs, high computational power, high memory bandwidth, and low power consumption. To verify if they are a viable solution, we have redesigned, implemented and optimized the LOFAR beam former for the NVIDIA GTX580 and the AMD HD6970 video cards, using the *Compute Unified Device Architecture* (CUDA) [22] and the *Open Computing Language* (OpenCL) [23] as implementation frameworks. OpenCL is a portable platform, and multi-core CPUs can also execute the OpenCL beam former. An important question is if OpenCL also provides performance portability. Therefore, on CPUs, we compare with an additional alternative implementation using OpenMP and manual SSE vectorization.

All versions exploit our auto-tuning and run-time code generation techniques to adapt the implementation to the hardware platform and problem parameters, as dictated by the observation specification. To gain insights in performance and power efficiency improvements, we compare our novel beam former with LOFAR's production version on the BG/P supercomputer, which is hand-written in assembly, and extremely efficient. Our results indicate that our auto-tuning many-core code is up to 50 times faster, and 3 times more power efficient on GPUs. Moreover, using 8 GPUs in a single node, our beam former even is 8 times more power efficient than the BG/P. This excellent result will allow us to build larger telescopes, and to point in more directions simultaneously, leading to more effective instruments.

The main research question addressed in this chapter is whether many-cores can be used to accelerate radio astronomy (see Section 1.1.2), as we study the feasibility of implementing and optimizing the LOFAR beam former on many-core GPUs and multi-core CPUs. Furthermore, we investigate the impact of auto-tuning on our parallel beam former, thus addressing the question of what is the impact of auto-tuning on radio astronomy algorithms (see Section 1.1.3), and by doing so we also provide results that are important to understand how difficult is auto-tuning many-core accelerators (see Section 1.1.5).

The rest of this chapter is organized as follows. First, we summarize the current state in the field of beam forming in Section 3.1. Then, we present background on telescopes, with a focus on LOFAR, and beam forming, in Sections 3.2 and 3.3 respectively. In Section 3.4 we analyze the LOFAR beam former, moving from the sequential to the GPU algorithm, while introducing the most important optimization strategies used. The auto-tuning of the beam former for the different architectures is introduced in Section 3.5. In Section 3.6 we present a detailed analysis of the algorithm's performance and power efficiency on different many-core platforms, while comparing to the production version. Finally, Section 3.7

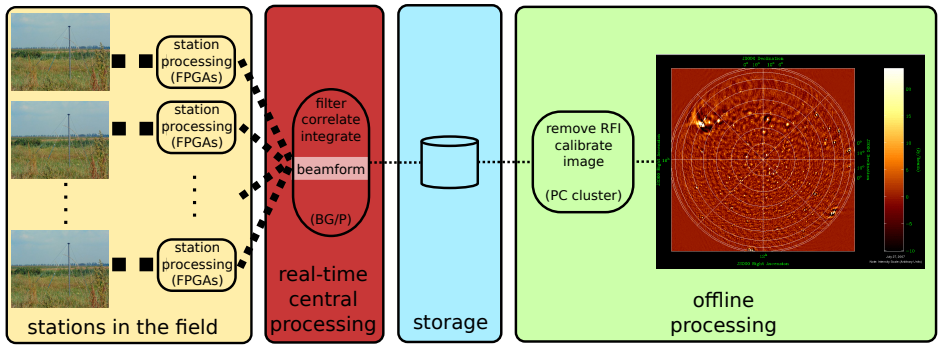

Figure 3.2: A simplified overview of LOFAR processing.

presents our conclusions.

3.1 Related Work

So far, there are few software beam formers in use in astronomy, as beam forming is an operation that is still implemented in hardware in the vast majority of cases. Among the software implementations, the one that is more relevant for our work is the LOFAR production beam former, described in [8]. An overview of the real-time software pipeline of LOFAR is presented in [7]. Another software beam former is the one of the *Giant Metrewave Radio Telescope* (GMRT) in India [24]. This beam former, running on a cluster of commodity hardware, produces at most 32 dual polarized beams, while the LOFAR beam former is capable of producing hundreds of different beams: this clearly shows how difficult the operational challenges of LOFAR are. It is interesting to observe that also for the GMRT the use of GPUs is considered as a possible future solution to improve performance and to cut costs [24].

A different approach to beam forming is the one of OSKAR [25]: a research tool, developed by the Oxford astrophysics and e-Research groups, used to investigate the challenges of beam forming for the SKA radio telescope. It supports two different modes of execution: the simulation of the beam forming phase and the computation of different beam patterns. Due to the fact that OSKAR is a simulation framework, it is not possible to directly compare its approach and performance with ours.

As far as we know, there are no GPU radio astronomy beam formers at the moment. There are, however, some attempts at implementing general domain beam formers using GPUs. Nilsen et al. [26] present two different digital beam

formers, implemented with CUDA, using an NVIDIA GeForce 8800. The authors conclude that they can see a future for the use of GPUs as the platform to run digital, high performance, beam formers. The hardware that we use in this chapter is significantly different, leading to different trade-offs. Also, our implementation is portable across different platforms thanks to the use of OpenCL and auto-tuning.

Beam forming is a general signal processing technique, and software beam formers are used in many different areas. For example, [27] presents two beam forming techniques aimed at increasing the number of possible users of an I-WiMAX maritime communication system. Beam forming is an efficient solution to reduce the spectrum necessary for wireless communication because the waves can be steered to the direction of the receiver, thus reducing interferences. Therefore, beam forming techniques are used in modern WIFI and 4G cellular telephone networks (TLE).

Another example of the importance of beam forming can be found in [28]. Here, the authors improve the flexibility of a radar system used to monitor the ocean, using a software beam former. The use of this kind of beam former permits to deploy this type of radar systems in locations that were not suitable before. Also, the field of medicine benefits from software beam forming. In [29], different beam forming algorithms are compared to find which one is the best for breast cancer detection using microwave imaging.

3.2 Software Telescopes

The structure of a classic radio telescope is relatively simple: first, a large metallic dish reflects the radio waves to an electronic receiver in its focal point. Next, the received signals are processed, usually by special-purpose hardware, and transformed into a representation that the astronomers can use. In the past, the need to improve precision and sensitivity of telescopes has always been addressed by building bigger dishes. However, this is a solution that does not scale: there are physical and engineering constraints on the dimensions that a dish may reach. In addition, moving a huge metal dish to point at some specific direction in the sky is slow and difficult. This is a severe limitation for several science cases, such as transient detection. Moreover, building a massive telescope of this kind implies enormous costs for raw materials and realization.

An attractive alternative is provided by radio interferometry, a technique that combines the signals received by different antennas into a single and meaningful signal. It is thus possible with this technique to use small dishes or antennas (simpler and cheaper than large metal dishes) and combine their measurements to build an instrument with a resolution that is equivalent to that of a telescope with a diameter equal to the largest distance between antennas. Such telescopes

essentially are distributed sensor networks.

However, a distributed telescope poses its own challenges: it is necessary to connect all receiving antennas to a central processing unit, and perform operations on the recorded samples to merge (correlate) them. Additionally, since the data streams are too large to store on disk, all these operations must be done in real-time. Implementing a system like this completely in hardware is costly and error prone, and may even span a period of over a decade to go from the design to the realization. Moreover, it requires expertises that are not easily found on the market. Most importantly though, hardware implementations lack flexibility: a telescope has a long lifetime (decades) during which there is a high probability that new requirements will arise, and that the operational setup will change. What appears to be the best solution for providing flexibility for the new generation of telescopes is to implement their operational pipelines in software.

In this chapter, we use the LOFAR telescope as a driving example. LOFAR is a large radio interferometric array, and a perfect example of what we call a software telescope: in 2012 it was the largest radio telescope in the world, with more than 80,000 antennas, and its software pipeline is executed in real-time by a two and a half rack IBM Blue Gene/P supercomputer. The antennas, all of them omni-directional, are of two different types: low-band antennas, for the frequency interval of 10-80 MHz, and high-band antennas, for the 110-240 MHz range. These antennas are not directly connected to the central computing facility, instead they are co-located in groups of different dimensions, and organized in *stations*. There are 20 core stations, all situated together in the northern part of the Netherlands, and 24 external stations at increasing distance from the core. Each station is equipped with a cabinet where some preliminary processing is performed. Each core station may act as two different stations, bringing the number of LOFAR stations to 64. Stations are important because they increase scalability: the software at the central computing facility can use the stations' output as its input, instead of dealing with each single antenna.

Figure 3.2 shows an overview of the LOFAR processing pipeline. The left part, before the storage, is executed in real time. This part includes the beam former. After beam forming, data is stored to disk, and processed further off-line. For more detailed information on the pipeline, we refer to [7]. In the next section, we will describe the beam forming process in more detail.

3.3 Beam Forming

Beam forming is a standard signal processing technique that is used to control the spatial selectivity of omni-directional antennas. It is important in radio interferometry, because the signals from the receiving antennas need to be compensated for the different antenna positions. This is shown in Figure 3.3: all antennas

receive the signals from a radio source at different moments in time. Signals are compensated before being integrated (added together), by shifting their phase and amplitude by a value that depends on the position of the source and of the antennas. Without this compensation, the formed beams would have no directionality. In case of narrow-band systems, such as LOFAR, the amplitude shift is unnecessary, and *a phase shift is sufficient to provide the correct compensation.*

LOFAR has two main software beam formers: the superstation and the tied-array beam former. The superstation beam former is part of the telescope's imaging pipeline. It reduces the number of stations seen by the correlator (i.e. the component that computes sky images [7]), thus lowering its complexity. This beam former simply adds samples together, without shifting them. We focus on the tied-array beam former, since it provides directionality in LOFAR.

The input samples of the beam former are grouped in *channels*. A channel represents an observation's frequency interval (in our case of 763 Hz). For this interval, the input contains all samples, for all used stations, measured in two polarizations (X and Y). The output consists of an arbitrary number of beams, that may vary between few and many hundreds, with each beam representing a different pointing direction. A beam contains all the frequency channels, and their samples in both polarizations. The complexity of the beam former is $O(s \cdot b)$, where s is the number of input stations and b is the number of generated output beams.

Beam forming is especially important for the detection of transient objects. Thanks to forming multiple beams at the same time, and thus pointing at many directions simultaneously, it is possible to use the telescope for monitoring a large fraction of the sky and wait for unexpected events. Moreover, if another instrument detects an object outside the area of the sky that LOFAR is currently monitoring, the software beam former can be reprogrammed to redirect its focus in real-time, without the need to move any mechanical part. We refer to [8] for more information on how beam forming is used in LOFAR.

3.4 Application Analysis

In this section, we first introduce the sequential beam forming algorithm. Subsequently, we present our multi-core and GPU parallel beam formers, explaining the parallelization and choice of optimization strategies.

3.4.1 The Sequential Algorithm

The LOFAR beam forming algorithm consists of three successive stages: (1) delays computation, (2) flagging bad samples, and (3) beam forming. The goal of the delays computation stage is to compute the time delays that are needed to

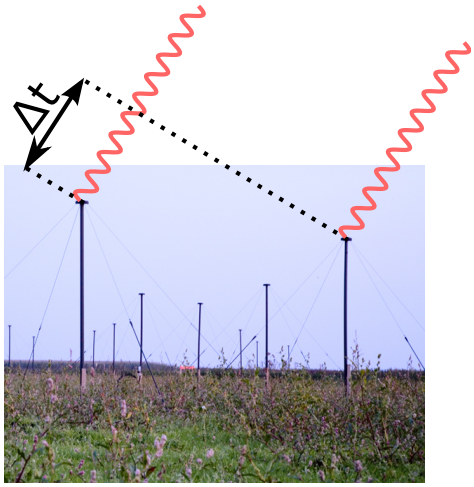

Figure 3.3: The rightmost antenna receives the signal earlier.

correct for the different antenna positions and for pointing the instrument. These are different for each possible combination of stations and beams. The input of this stage are the delays at the beginning and the end of a samples period, for each pair of stations and beams. For each beam and station the algorithm computes an average of these two values, subtracts from it the precomputed delay of the first station, and stores the result in a matrix.

The second stage, flagging bad samples, is necessary because received signals may contain errors due to radio frequency interference (RFI). Since the beam former integrates data from different stations, errors from one station will propagate, and pollute all output beams. A station is invalid and excluded from the computation if the number of its flagged samples, i.e. the samples containing errors, is above a certain threshold. Even if a station is valid it may still contain some errors. To account for this, if a station's sample is flagged, the corresponding sample in all output beams is flagged as well.

The third stage is the proper beam forming stage, and it accounts for most of the execution time. For each beam to form, the algorithm iterates over all three dimensions (channels, samples and polarizations), loads each station's samples, phase shifts them, and outputs the average of these shifted samples. Algorithm 1 shows pseudocode for the main loop of this stage. The phase shift value is a function of the previously computed delay, and of the sample's frequency. All the signals are represented as single precision floating point complex numbers, so all the arithmetic operations in the pseudocode are in fact complex operations.

The delays computed in the first stage are used only to compute the phase

Algorithm 1 Pseudocode for the third stage of the beam forming algorithm.

```
for b = 0 → nrBeams do
  for c = 0 → nrChannels do
    for t = 0 → nrTimes do
      for p = 0 → nrPolarizations do
        beam = 0
        for s = 0 → nrStations do
          sample = Input[c][s][t][p]
          shift = computeShift(c, s, b)
          sample = sample * shift
          beam = beam + sample
        end for
        Output[b][c][t][p] = beam/nrValidStations
      end for
    end for
  end for
end for
```

shifts in the third stage. Therefore, we extract the phase shifts computation from the third stage and move it to the first. Thus, the *computeShift()* operation in Algorithm 1 simply becomes an access to a lookup table. As a side effect of this merge, the beam forming stage is simplified, and uses only single-precision floating point operations. In fact, the only operations now are complex additions and multiplications, which are efficiently implemented on all platforms. Even after the removal of the phase shifts computation, the third stage remains the most time consuming part of the algorithm.

Analysis of the sequential algorithm shows that the beam forming algorithm is inherently parallel: there are no data dependencies between different beams, and they can be computed independently of each other. We can affirm that the delays computation and the flagging stages don't have dependencies between them, thus they can be independently computed, possibly concurrently. The third stage, however, has to wait until the completion of the previous stages, as its computation depends on their outputs.

3.4.2 The IBM Blue Gene/P Production Version

The production version of the LOFAR beam former, in 2012, ran on an IBM Blue Gene/P supercomputer. It is implemented in C++, with the core routines written in assembly for performance reasons. The code presented in Algorithm 1 is written in assembly and is manually tuned to minimize memory accesses and

maximize hardware utilization. This highly optimized implementation of the beam former achieves 86% of the platform's peak performance. For a further description and performance analysis of the production beam former we refer to [8].

3.4.3 The Multi-core CPU Version

A beam former for multi-core CPUs needs to be flexible enough to leverage the ever increasing number of available cores per node. Moreover, modern CPUs provide different levels of parallelism as they also support SIMD instructions (SSE); exploiting all these levels of parallelism at the same time is critical to achieve high performance.

Given that the phase shifts are computed for each combination of channels, stations and beams, and that each phase shift is independent of all the others, we use OpenMP [30] to divide the work in the first stage between different threads. For each different channel a thread works on a non-overlapping subset of stations, and computes the phase shifts associated with these stations and all the beams.

In the flagging stage, we can use OpenMP to parallelize counting the valid stations and flagging the output. However, this part of the algorithm is less computation intensive and, consequently, benefits less from parallelization. Therefore, we do not further investigate the parallelization of this stage.

For stage three, we have two different levels of parallelism: (1) samples are equally divided between OpenMP threads and (2) the two polarizations of each sample are computed in parallel using the Streaming SIMD Extensions (SSE) [31]. The beam former kernel uses a different thread for each frequency channel and, for each channel, this thread spans a small number of children, each of them working on a part of the samples and being responsible for the computation of all beams, i.e. it merges all the shifted samples, in both polarizations, of all the different stations.

3.4.4 The GPU Version

Here we introduce the parallelization strategies behind our GPU algorithm; as common in high performance computing, these strategies are a direct consequence of the hardware organization of the platform. The described algorithm is the best performing one of a family of six different GPU beam forming algorithms that we designed and implemented in previous work [12]. To achieve good performance it is necessary to understand that GPUs are inherently hierarchical devices. In fact, they use two different hierarchical abstractions: the computational and the memory organization. From a computational point of view, GPUs are equipped with a variable number of streaming multiprocessors, each of which contains many computational cores. GPU computations are organized in thread-blocks and threads,

with each thread-block being associated with a streaming multiprocessor, and each of the block's threads being executed by one of the streaming multiprocessor's cores. In addition, GPUs have several different memories. Most important are the off-chip global memory, accessible by all threads of all thread-blocks, and the on-chip shared memory, that is available only to the threads of a same block, and may be used as an application-controlled cache.

Data transfers from the host to the GPU over the PCI-e bus can be a bottleneck for data-intensive computations [32]. In radio astronomy signal processing, this has also been identified as a problem [4]. However, for this work, we do not take the host-GPU data transfers into account, since the beam former is a part of a larger pipeline [7]. Therefore, we assume the data already is on the GPU, and may also be used on the GPU again for further processing.

The delays and phase shift computations present a degree of parallelism that may benefit from a GPU implementation. However, they use double-precision floating point operations and trigonometric functions, both of which are expensive on GPUs. We did investigate GPU parallelization of this stage, but the OpenMP/SSE CPU implementation performed better.

The flagging stage has less advantage of being parallelized on the GPU, since it has limited data parallelism and uses non-trivial data structures (i.e. C++ sparse sets). Moreover, of this stage's outputs, only the number of valid stations is directly used in the following phase, and this value can be easily passed as an argument to the GPU kernel.

The third phase of the algorithm benefits the most from parallelization on a GPU. In the design of our GPU beam former we exploit two levels of parallelism. In the first level, we assign each channel to a different thread-block. All samples are independent in the time direction, thus inside each thread-block we assign a different sample to every single thread. Each thread is then responsible for merging and shifting all stations and all beams, in both polarizations, but just for the channel and time associated with it. The advantage of this solution is that it eliminates the need for any inter thread, and inter thread-block communication: each thread can run independently from the others.

Moreover, this structure for the computation permits coalesced accesses to the GPU's global memory, as the threads inside a block read their inputs from, and write their outputs to, consecutive memory addresses. This is important because, on GPUs, coalescing is critical for performance [33]. Due to hardware or implementation framework limitations on some platforms the structure may be slightly modified at runtime, i.e. splitting each original thread-block into multiple blocks, but this does not modify the overall structure of the algorithm.

3.4.5 The beams-block Optimization Strategy

Sections 3.4.3 and 3.4.4 outlined how the beam forming algorithm can be parallelized. However, achieving good performance is still difficult. To achieve good performance on architectures where the gap between computational power and memory bandwidth is wide (such as on GPUs), it is extremely important to minimize the number of accesses to the slow global memory [4]. Minimizing memory accesses is even more important for algorithms with low arithmetic intensity (AI) [19], such as the beam former. Its arithmetic intensity, the number of operations performed per byte accessed in global memory, can simply be counted looking at the source code, and is shown in Equation 3.1.

$$AI = \frac{(4 \times stations) + 1}{(6 \times stations) + 4} \tag{3.1}$$

A way to minimize accesses to memory is to increase data reuse: when loading data from global memory, it is important to perform all, or most of, the operations associated with these data. There are two different points in our algorithm with potential for data reuse: (1) all the threads of a thread-block may share the phase shifts, because they are independent with respect to time, and (2) a single thread may reuse a loaded station's sample to compute many beams. To implement this optimization strategy we modified the kernel's main loop as shown in Algorithm 2.

Algorithm 2 Pseudocode for the kernel's beams-block optimization.

$c = myChannel()$
$t = myTime()$
for $station = firstStation \rightarrow nrStations$ **do**
  $samplePolarization0 = Input[c][station][t][0]$
  $samplePolarization1 = Input[c][station][t][1]$
  for $beam = firstBeam \rightarrow firstBeam + beamBlockDim$ **do**
    $shift = Shifts[c][station][beam]$
    $beam0 = samplePolarization0 * shift$
    $beam1 = samplePolarization1 * shift$
    $Beams[beam][0] = Beams[beam][0] + beam0$
    $Beams[beam][1] = Beams[beam][1] + beam1$
  end for
end for

The first crucial difference with Algorithm 1 is that the order of the loops over stations and beams is changed. This enables reuse of loaded station samples to form many beams. Another important difference is that the beams loop is not over the whole space of the beams, but only over a block.

Platform	Cores	GFLOP/s	GB/s	TDP (Watt)
IBM Blue Gene/P	4×1	13	13	24
Intel Xeon E5620	4×2	153	51	160
AMD Opteron 6172	12×4	806	170	320
AMD HD6970	1536	2793	176	250
NVIDIA GTX580	512	1581	192	244

Table 3.1: Characteristics of the used platforms.

This is necessary because it is not always possible to compute all the beams within a single kernel execution due to the limited number of registers. Even though we use arrays in the pseudocode, the actual code uses registers instead of memory for performance reasons. Thus, the number of beams that can be formed within a single execution is limited by the number of registers that are available per thread. Furthermore, in the source code the memory operations are vectorized, so the two polarizations are loaded and stored with a single operation.

We call the number of beams formed during an iteration of the innermost loop the *beams-block*. This parameter is of capital importance for the performance of the algorithm: a correct setup may effectively improve performance, while a wrong one may lead to hardware underutilization, a non-optimal number of memory accesses if the block is too small, or register spilling if the block is too large. The performance improvement brought by the beams-block optimization strategy is also reflected in the arithmetic intensity, as shown in Equation 3.2. The kernel's arithmetic intensity increases with a larger beams-block size.

$$AI = \frac{(4 \times stations) + 1}{\frac{4 \times stations}{beams-block} + (2 \times stations) + 4} \tag{3.2}$$

3.5 Auto-tuning the Beams-block

To provide a fair platform performance comparison, we need to select the best configuration of the beams-block parameter. To tune the algorithm, we try different configurations, and measure the number of single precision floating point operations per second (GFLOP/s) achieved in the third stage. The analysis of these values provides general and case-specific guidelines for the setup of the beams-block.

The experiments are performed using the *Distributed ASCI Supercomputer 4* (DAS-4) [34], running CentOS Linux release 6. The C++ compiler used for the CPU code is the GNU g++, version 4.4.6, that implements OpenMP 3.1; we

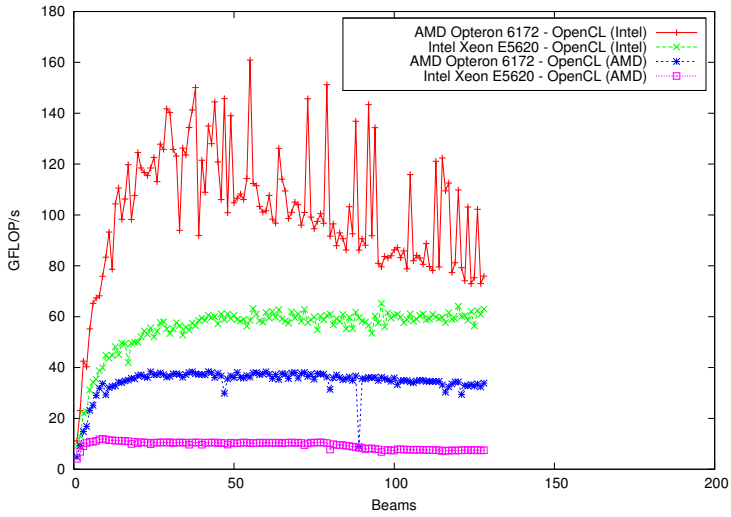

Figure 3.4: Comparison of Intel and AMD OpenCL compilers for CPU, 512 stations (higher is better).

obtained slightly lower results accross the board using the Intel C++ compiler, version 12.1, thus we are not including it in the discussion. For running the OpenCL implementation on multi-core CPUs we rely on the runtime environment included with the Intel OpenCL SDK 1.1, as we found that for our application it is much faster than AMD's SDK (see Figure 3.4). For the NVIDIA GPU we use CUDA 4.0, that also provides an OpenCL runtime, while for the AMD GPU we use the *AMD Accelerated Parallel Processing* (APP) SDK version 2.5. Platform characteristics are summarized in Table 3.1.

3.5.1 IBM Blue Gene/P

The LOFAR production code uses an optimization strategy based on combining multiple iterations of the beam former's main loop. After careful analysis and manual tuning we found that the optimal setup is to compute 128 time samples, 6 stations and 3 beams for each kernel iteration. The tuning of the production version of the beam former on the BG/P is beyond the scope of this work, thus we do not further discuss it and refer to [8] for more information.

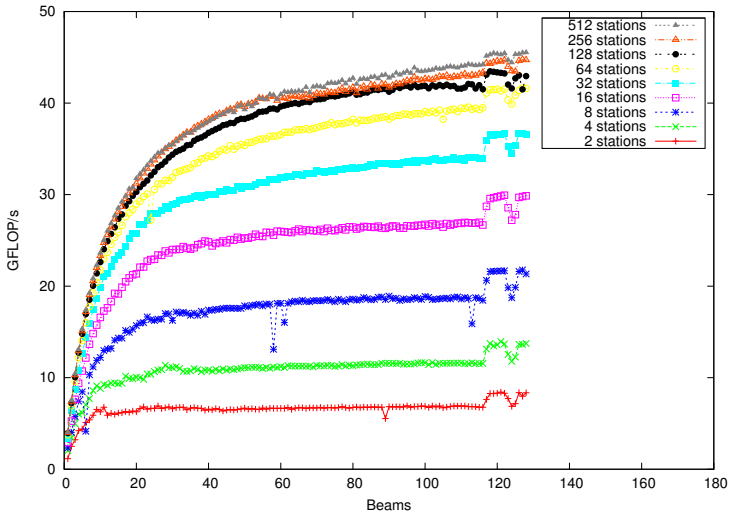

Figure 3.5: Tuning the beams-block for the Intel Xeon E5620 using OpenMP/SSE (higher is better).

3.5.2 Intel Xeon E5620

The first multi-core CPU we use is the Intel Xeon E5620. Figure 3.5 presents the results obtained using the OpenMP/SSE version (described in Section 3.4.3). Results show that performance is increasing with the size of the beams-block. The returns are diminishing for higher values, however, because the beams-block size determines the number of registers used, and for higher values, the compiler spills the registers to memory. Nevertheless, the best setup for this platform is to set the beams-block equal to the number of beams to form.

The Intel CPUs also support OpenCL, and we auto-tuned the OpenCL implementation (described in Section 3.4.4) for this platform as well. Figure 3.6 shows the results for this experiment. Despite some outliers, the behavior of the OpenCL implementation is close to the OpenMP/SSE version. With OpenCL, the best setup of the beams-block is also to use a value equal to the number of beams.

Since we have two different implementations on the same hardware, we compare the performance achieved on this platform by OpenMP/SSE and OpenCL in Figure 3.7. The comparison shows that the OpenCL implementation achieves higher performance than the OpenMP/SSE version: the OpenCL implementations achieves 65 GFLOP/s (42% of the platform's peak), while the OpenMP/SSE implementation achieves only 45 GFLOP/s (29% of the platform's peak). This

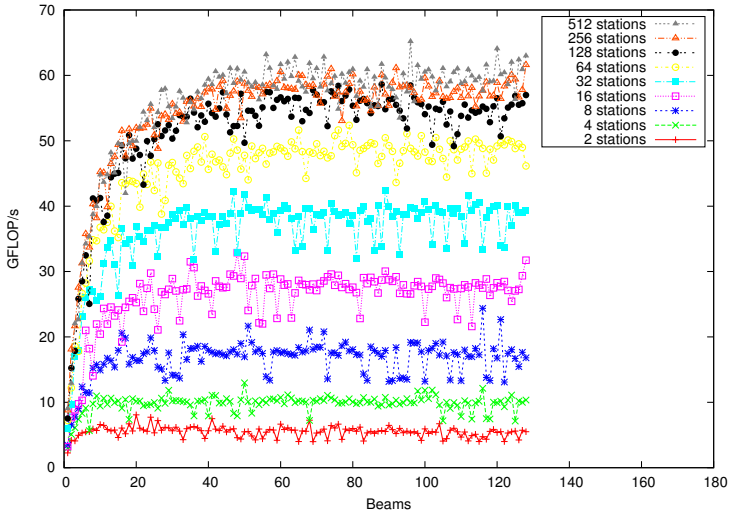

Figure 3.6: Tuning the beams-block for the Intel Xeon E5620 using OpenCL (higher is better).

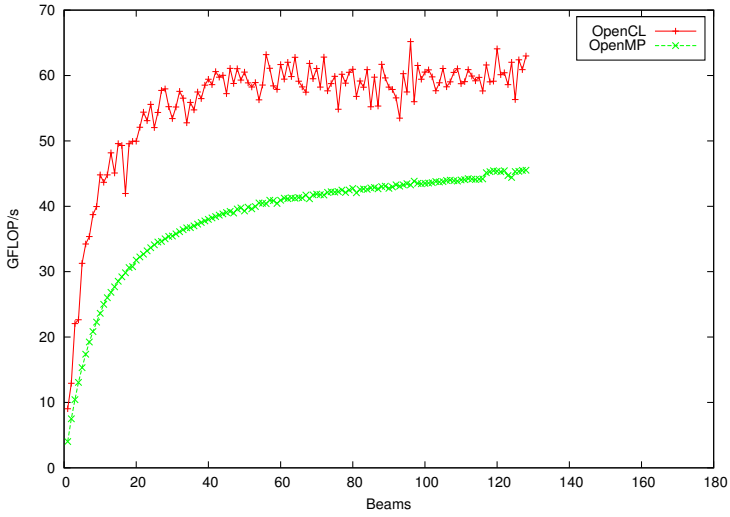

Figure 3.7: Comparison between OpenMP/SSE and OpenCL on the Intel Xeon E5620, 512 stations (higher is better).

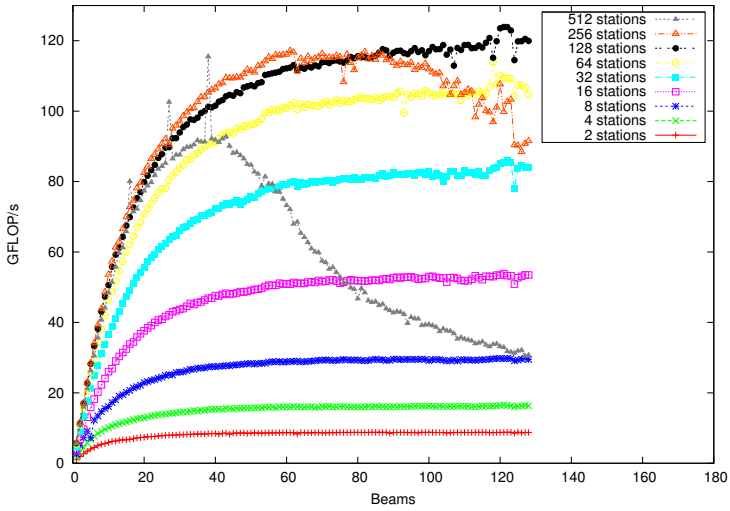

Figure 3.8: Tuning the beams-block for the AMD Opteron 6172 using OpenMP/SSE (higher is better).

result can be explained by further optimizations applied by the Intel's OpenCL compiler compared to GCC.

3.5.3 AMD Opteron 6172

The second multi-core CPU that we use is the AMD Opteron 6172. We run the same experiment that we previously described for the Intel CPU. Figures 3.8 and 3.9 present the results obtained using OpenMP/SSE and OpenCL, respectively.

The OpenMP/SSE implementation behaves almost the same as on the Intel CPU, with the difference that on the Opteron CPU the curves are plateauing later, thanks to the higher number of available cores. We also see that for a high number of stations performance is not always increasing with the beams-block size, i.e. it rises to a peak and then there is a sudden drop. Thus, we can conclude that the best setup of the beams-block for this platform equal to the number of beams when there are not so many stations, and to select a smaller value if the number of stations is larger than 128. The specific value is input dependent and can be identified using the results of this auto-tuning step.

The behavior of the OpenCL implementation on the Opteron 6172 is far less stable than what observed on the Intel CPU, and even less stable than the OpenMP/SSE implementation on the same hardware. However, if we exclude the

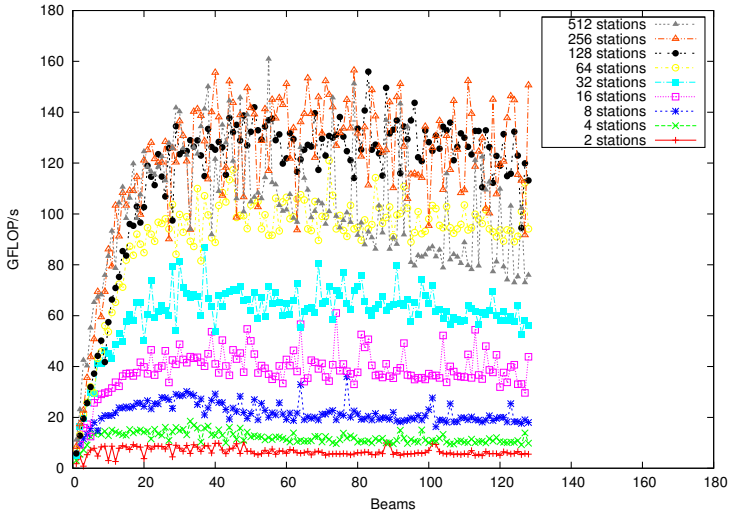

Figure 3.9: Tuning the beams-block for the AMD Opteron 6172 using OpenCL (higher is better).

outliers, we see the same trend that we had with the OpenMP/SSE implementation, thus we can conclude the same for what concerns the size of the beams-block size.

Figure 3.10 shows a comparison of OpenMP/SSE and OpenCL for this platform. The OpenCL implementation again achieves more GFLOP/s than OpenMP/SSE: 161 against 123. However, our many-core beam former is less efficient on the AMD CPU, reaching only 20% and 15% of the theoretical peak, respectively. This is caused by the relatively low memory bandwidth *per core* of the AMD machine, which hurts our data-intensive code.

3.5.4 NVIDIA GTX580

The NVIDIA GTX580 card can run both CUDA and OpenCL. The number of stations varies between 2 and 512, while the number of beams varies between 1 and 16; we have this large difference in the maximum number of stations and beams because the number of stations is limited only by the available global memory (3 GB in our configuration). The number of beams, however, is limited by the number of available registers, and we hit the ceiling of 63 registers per thread (a hardware property of the NVIDIA Fermi architecture) before the value of 16 for the beams-block. Larger numbers of beams are computed by using multiple blocks. The results using CUDA are presented in Figure 3.11.

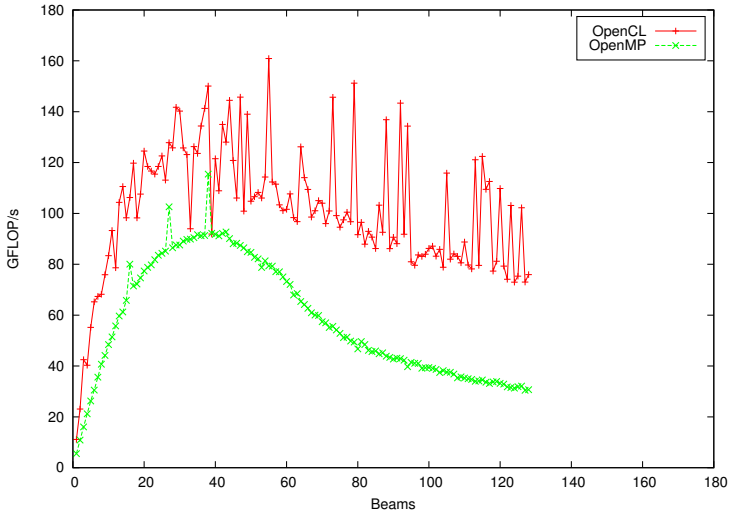

Figure 3.10: Comparison between OpenMP/SSE and OpenCL on the AMD Opteron 6172, 512 stations (higher is better).

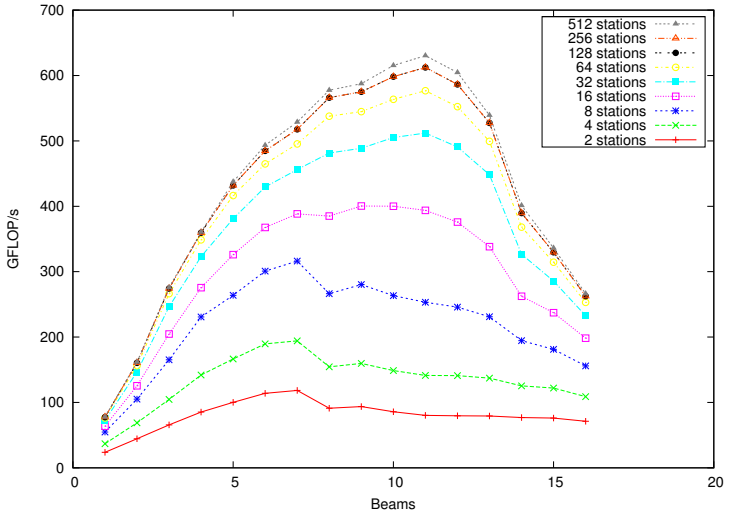

Figure 3.11: Tuning the beams-block for the NVIDIA GTX580 using CUDA (higher is better).

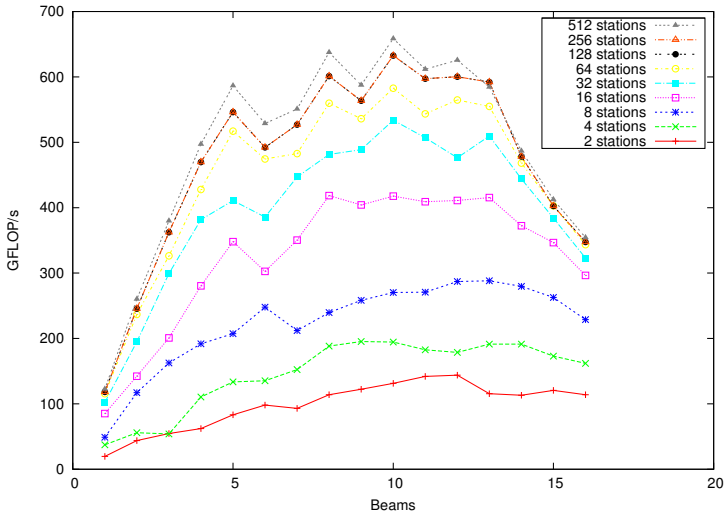

Figure 3.12: Tuning the beams-block for the NVIDIA GTX580 using OpenCL (higher is better).

The behavior appears to be regular, with performance increasing with the growing of the beams-block. We can see that for few stations, the performance's peak is found in correspondence of the value of 7, while for a large number of stations the same peak is found in correspondence of a beams-block of 11. We suspect that these performance peaks are caused by the cache behavior. We also show that for values larger than 11 beams, there is a sudden loss in performance, due to register spilling. The highest achieved number of GFLOP/s for the GPU beam former is 642, 40% of the card's theoretical peak performance. This is an excellent result for an algorithm that is as data intensive as the beam former.

The OpenCL implementation produces results that are comparable, as shown in Figure 3.12. We see, however, that the OpenCL behavior is more irregular: instead of smooth curves we have spiked ones. Differently from the CUDA implementation, the performance's peak appears to be in correspondence with a greater value of the beams-block for a little number of stations, and to retreat back to the value of 10 for a high number of stations. Also the decrease in performance with large beams-block sizes is less steep.

We present the difference between OpenCL and CUDA with 512 stations in Figure 3.13. Overall, the OpenCL implementation provides slightly higher performance, reaching a peak of 672 GFLOP/s (42% of the theoretical peak), but the difference with the CUDA implementation is less than 2%. It is more

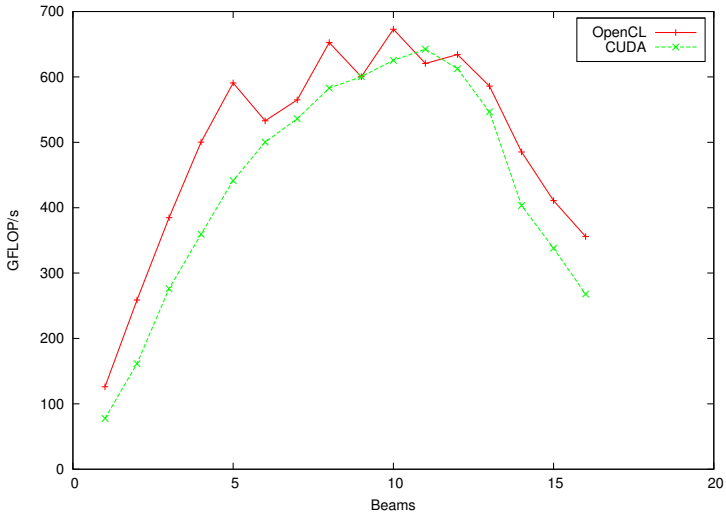

Figure 3.13: Comparison between CUDA and OpenCL on the NVIDIA GTX580, 512 stations (higher is better).

difficult to tune the beams-block for the NVIDIA GTX580 than it was for the multi-core CPUs. In general we can affirm that a value higher than 11 for the CUDA implementation, or 13 for the OpenCL implementation, badly affects performance. However, the best value of the beams-block is so dependent on the particular input size that a decision should be deferred until this value is known. When known, it can be easily computed thanks to the results of our auto-tuning step. More than for multi-core CPUs the auto-tuning looks of capital importance to achieve good performance with GPUs, as we can see that the performance window of these architectures is small.

3.5.5 AMD HD6970

The second GPU platform is the AMD HD6970, on which we run the OpenCL implementation of the beam former. For this platform we use different values for stations and beams: the number of stations varies between 2 and 64, and the number of beams between 1 and 51. These numbers reflect two large differences between this card and the NVIDIA GTX580: first, the HD6970 imposes limits on the amount of memory that can be allocated in a single block, thus reducing the maximum number of stations that is possible to merge within a single run; second, the HD6970 provides more registers per thread, thus increasing the beams-block space. This is caused by a difference in GPU architecture: AMD GPUs have a

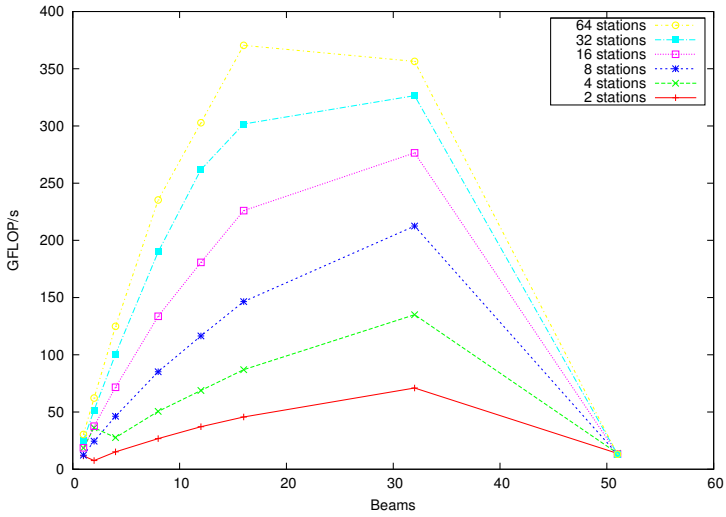

Figure 3.14: Tuning the beams-block for the AMD HD6970 using OpenCL (higher is better).

set of registers per core, while on NVIDIA GPUs, the register file is shared by all cores in a streaming multiprocessor. The results are presented in Figure 3.14. For this platform we test only the OpenCL implementation, without providing a comparison with the AMD native GPU framework.

It is interesting to observe that peaks are not confined to the beginning of the space: there are peaks at as high as a beams-block of 32, thanks to the large number of registers of this platform. Also the behavior is more stable than on the NVIDIA GPU, and the general advice for this platform is to use a larger value for the beams-block. Unfortunately, the impossibility to merge more than 64 stations in a single kernel execution prevents this platform to achieve performance comparable with the GTX580: we measure a peak performance of 370 GFLOP/s, only the 13% of the GPU's theoretical peak.

To conclude, Table 3.2 presents a summary of the highest measured GFLOP/s for all the platforms, together with the achieved efficiency. We see that our GPU beam former, compared with the production implementation, provides a performance improvement, per chip, of more than 60 times, even though the efficiency is lower due to the wide gap that GPUs have between memory bandwidth and computational power.

Platform	GFLOP/s	Efficiency
IBM Blue Gene/P	10	79%
Intel Xeon E5620 (OpenMP/SSE)	45	29%
Intel Xeon E5620 (OpenCL)	65	42%
AMD Opteron 6172 (OpenMP/SSE)	123	15%
AMD Opteron 6172 (OpenCL)	161	20%
AMD HD6970 (OpenCL)	370	13%
NVIDIA GTX580 (CUDA)	642	40%
NVIDIA GTX580 (OpenCL)	672	42%

Table 3.2: Maximum single run performance achieved during the auto-tuning.

3.6 Performance Analysis

In this Section we describe the performance and power efficiency results obtained by our many-core beam former, and compare the results of the different architectures with the production implementation.

3.6.1 Performance comparison for a sky survey observation

To answer the question if many-core architectures are suitable to accelerate beam forming in radio astronomy, we compare the performance of our beam former running on all the architectures presented in Table 3.1 with the production one, using a real use case from the LOFAR telescope.

For this experiment, the number of input stations is 64, the highest available in the LOFAR operational setup, with each station providing one second of observation, divided into 768 dual polarized samples, covering a frequency spectrum composed of 256 different channels. The number of beams to form as output is 155. This configuration is typical for a *sky survey observation*. The beams-block is set according to the results of Section 3.5. The achieved GFLOP/s, and the speedups relative to the Blue Gene/P implementation, are available in columns 3 and 4 of Table 3.3.

In terms of raw computational power the GPUs are clear winners: our many-core beam former running on GPUs achieves 5–14 times more single precision floating point operations per second than on multi-core CPUs. The best performing platform is the AMD HD6970 GPU, achieving 612 GFLOP/s, 7% more than the same OpenCL implementation running on the NVIDIA GTX580. For what concerns the GTX580, we measure a difference of nearly 4% between the CUDA and OpenCL implementations, with the latter performing slightly better

Platform	AI	GFLOP/s	Speedup	Watt	GFLOP/Watt
IBM Blue Gene/P	-	12	1	24	0.45
Intel Xeon E5620 (OpenMP/SSE)	1.92	42	3.4	360	0.11
Intel Xeon E5620 (OpenCL)	1.92	49	4.0	342	0.14
AMD Opteron 6172 (OpenMP/SSE)	1.92	106	8.7	625	0.16
AMD Opteron 6172 (OpenCL)	1.92	88	7.3	535	0.16
AMD HD6970 (OpenCL)	1.73	612	50.6	439	1.39
NVIDIA GTX580 (CUDA)	1.65	552	45.6	467	1.18
NVIDIA GTX580 (OpenCL)	1.63	572	47.3	455	1.25

Table 3.3: Platform comparison for a typical sky survey observation, merging 64 stations into 155 beams.

than the former. It is important to note that, thanks to auto-tuning, our beam former shows performance portability between the different GPUs.

For what concerns the two multi-core CPUs, the OpenCL implementation provides higher performance than OpenMP/SSE on the Intel CPU, while on the AMD CPU it is the OpenMP/SSE implementation that performs better.

The IBM Blue Gene/P achieves 12.1 GFLOP/s in this configuration, 88% of the theoretical peak. In terms of efficiency this platform remains the best performing one, but, due to its design, it scores lowest in terms of achieved GFLOP/s. Comparing the results of our many-core beam former with the production implementation, we measure an improvement of 3–8 times for multi-core CPUs and of 45–50 times for GPUs.

3.6.2 Power efficiency for a sky survey observation

Our GPU beam former provides high performance, as we demonstrated in Section 3.6.1, and can be tuned to different platforms and observation modes, as shown in Section 3.5. However, this is not enough: future software telescopes require high performance, but must also be highly power efficient. For LOFAR, power dissipation already accounts for a large part of the instrument's operational costs. For future telescopes, this will likely be even worse. When evaluating the computational architectures of future telescopes, it is thus necessary to look for an architecture that will maximize the number of operations that is possible to provide *per Watt of consumed power*. Therefore, we measure the power efficiency of our beam former for the different platforms we evaluate, again comparing with the production version on the Blue Gene/P. To measure power consumption, we run the different beam formers in the same operational environment (the sky survey observation) as described in Section 3.6.1, and use the DAS-4 Schleifenbauer Power Distribution Units (PDUs) to measure the power dissipated by the working nodes. The measurements are provided in Table 3.3.

The IBM Blue Gene/P uses the least power per chip in absolute sense, as expected, since it is designed for low power consumption. However, the architecture also (by design) has a relatively low floating point performance per chip: it provides nearly half a GFLOP for each consumed Watt of power. This is not surprising, as the chip is created with an older manufacturing process (90 nm). Moreover, the Blue Gene/P also contains hardware for five different networks. These results are better than those achieved by the multi-core CPUs that we tested: they provide only 0.11–0.16 GFLOP per Watt. In contrast to the Blue Gene/P, however, these CPUs are focused on single core performance and not on power efficiency.

With 1.18–1.39 GFLOP per Watt, the GPUs are 2–3 times more power efficient than the Blue Gene/P, and 7–12 times more efficient than the multi-core

CPUs we tested. Moreover, the GPU power efficiency increases when using more GPUs per node, since the power budget of the host is spread over more GPUs and more FLOPs. In fact, we experimented with a special DAS-4 node containing 8 NVIDIA GTX580 GPUs, resulting in a GFLOP per Watt ratio of 3.72. This is 8 times more efficient than the Blue Gene/P, and 23–33 times more than the multi-core CPUs.

3.7 Conclusions

Radio telescopes are quickly changing into gigantic sensor networks with complex real-time software pipelines. To efficiently implement future telescopes with exascale performance demands, we need to evaluate computing platforms that can provide high performance, while simultaneously being highly energy efficient. In this chapter, we evaluated the beam forming algorithm, an important radio telescope building block, but also used in computer networks, radar systems and medical equipment. We evaluated this algorithm on seven many-core hardware and software platform combinations, while comparing to the production version of the beam former of LOFAR, one of the largest radio telescopes in the world, which used a Blue Gene/P supercomputer.

Optimizing memory access is of capital importance when dealing with platforms where the gap between computational power and memory bandwidth is wide, as it is on GPUs. This problem becomes increasingly important with virtually all modern architectures, as the number of compute cores grows much faster than the memory bandwidth. This is especially difficult with data-intensive algorithms, such as the beam former, which has a low arithmetic intensity.

We parallelized the algorithm for both multi-core CPUs and modern many-core GPUs. We implemented our solutions using OpenMP with SSE vector instructions, CUDA and OpenCL. To achieve high performance on these architectures, we modified the sequential algorithm and implemented many optimization techniques aimed at minimizing memory accesses. We maximized data reuse at different levels, both inter- and intra-thread, while at the same time ensuring that the algorithm uses coalesced access to memory.

Since many-core platforms are changing rapidly, and telescopes have lifetimes of decades, we do not focus on a specific many-core architecture, but aim to be portable, in terms of both code and performance. We demonstrated that it is possible to use auto-tuning to achieve high performance for different combinations of hardware platforms and implementation frameworks. We use run-time compilation techniques to automatically tune the code for a particular input problem (observation specification) and hardware platform dynamically. We showed that performance portability is possible in practice: our auto-tuned OpenCL code achieves the same or better performance than hand-optimized code, on both

GPUs and multi-core CPUs. Moreover, we believe that this approach to code and performance portability, based on run-time code generation and auto-tuning, may be applied to different parallel applications for many-core architectures. We are now applying this same approach to other radio astronomy algorithms.

Our approach leads to exciting results: compared to the production implementation, our auto-tuned beam former is 45–50 times faster and 2–3 times more power efficient on GPUs, and even 8 times more power efficient when using 8 GPUs in a single node. Furthermore, the GPU solution remains the fastest even when taking the host-GPU memory transfers into account. We conclude that GPUs provide a viable solution for high performance and energy efficient beam forming in radio astronomy, and thus provide a first positive answer for **RQ1** (see Section 1.1.2).

Chapter 4

Dedispersion*

Some astronomical sources, such as pulsars, emit millisecond duration, impulsive signals over a wide range of radio frequencies. As this electromagnetic wave propagates through the ionized material between us and the source, it is dispersed. This causes lower radio frequencies to arrive progressively later and without correction this results in a loss of signal-to-noise that often makes the source undetectable when integrating over a wide observing bandwidth. For a fixed interval of frequencies, this dispersion is a non-linear function of the distance between the emitting source and the receiver, that can be reversed by simply shifting in time the signal's lower frequencies. This process is called *dedispersion*. Dedispersion is a basic algorithm in high-time-resolution radio astronomy, and one of the building blocks of modern radio telescopes like the Low Frequency Array (LOFAR) and the Square Kilometer Array (SKA). The amount of processing needed for dedispersion varies per instrument, but can be in the petaflop range. Hence, it is important to have a high performance, adaptable and portable dedispersion algorithm.

Due to the nature of the problem, however, designing a high performance dedispersion algorithm is far from trivial. In fact, dispersion can be easily reversed if the distance of the source from the receiver is known in advance, but this is not true when searching for unknown objects in surveys for pulsars or fast-transient sources. When searching for these celestial objects, the distance is one of the unknowns, and the received signal must be dedispersed for thousands of possible trial distances. This results in a brute-force search that produces many dedispersed signals, one for each trial distance. Clearly, this search is both computationally and data intensive, and, due to the extremely high data-rate

*This chapter has been adapted from "Auto-Tuning Dedispersion for Many-Core Accelerators" [13]

of modern radio telescopes, it must be performed in real-time, since the data streams are too large to store in memory or on disk. Luckily, all these different searches are independent from each other, and can be performed in parallel.

We aim to achieve high performance by parallelizing this algorithm for many-core accelerators. Recently, similar attempts have been made by Barsdell et al. [35] and Armour et al. [36]. However, we believe that the performance analysis presented there is not complete. Moreover, the focus in [35] and [36] is on specific platforms and observational setups, while in this chapter we focus on designing a portable many-core algorithm that can be tuned for different platforms and, more importantly, different radio telescopes and observational setups. To our knowledge, this is the first attempt at designing a dedispersion algorithm that is not fine tuned for a specific platform or telescope. Furthermore, even if dedispersion is an inherently parallel algorithm, it is still interesting as it represents a class of applications that, due to their low arithmetic and high data intensity, is not often implemented on accelerators. We believe that these applications do not only push the limit of many-core architectures, but can also benefit from the higher memory bandwidth that most many-cores provide, compared with traditional CPUs.

We designed and developed a many-core dedispersion algorithm, and implemented it using the Open Computing Language (OpenCL). Because of its low arithmetic intensity, we designed the algorithm in a way that exposes the parameters controlling the amount of parallelism and possible data-reuse. In this chapter we show how, by auto-tuning these user-controlled parameters, it is possible to achieve high performance on different many-core accelerators, including one AMD GPU (HD7970), three NVIDIA GPUs (GTX 680, K20 and GTX Titan) and the Intel Xeon Phi. We not only auto-tune the algorithm for different accelerators, but also use auto-tuning to adapt the algorithm to different observational configurations. In radio astronomy, observational parameters are more variable than the platforms used to run the software, so being able to adapt the algorithm to different observational setups is of major importance. Furthermore, in this work we measure how much faster a tuned algorithm is compared to every other possible configuration of the parameters, and quantify the statistical difference between optimum and average performance. Finally, with this work we are able to provide a comparison of modern accelerators based on a real scientific application instead of synthetic benchmarks.

To summarize our contributions, in this chapter we: (1) provide an in-depth analysis of the arithmetic intensity of dedispersion, providing analytical evidence and empirical proofs of it being memory-bound, in contrast to earlier claims in the literature; (2) show that by using auto-tuning it is possible to adapt the algorithm to different platforms, telescopes, and observational setups; (3) demonstrate that it is possible to achieve real-time performance using many-core accelerators; (4)

quantify the impact that auto-tuning has on performance; and (5) compare different platforms using a real-world scientific application.

The main research question addressed in this chapter is whether many-cores can be used to accelerate radio astronomy (see Section 1.1.2), and we do so designing, implementing, and optimizing a parallel and dedispersion algorithm. Furthermore, we investigate the impact of auto-tuning on our dedispersion algorithm, thus addressing the question of what is the impact of auto-tuning on radio astronomy algorithms (see Section 1.1.3). While showing the impact that auto-tuning has on performance, and portability, of dedispersion, we also look at how difficult is auto-tuning of many-core accelerators (see Section 1.1.5).

The subsequent parts of this chapter are organized as follows. Section 4.2 provides a brief introduction to the theory of dedispersion and the challenges associated with it. Section 4.3 presents the dedispersion algorithm, introducing our parallel implementation and its optimizations; this section includes the theoretical analysis of dedispersion's arithmetic intensity. The experiments used to validate our work are described in Section 4.4, while their results are presented in Section 4.5. Finally, relevant literature is discussed in Section 4.1, and Section 4.6 summarizes our conclusions.

4.1 Related Work

In the literature, auto-tuning is considered a viable technique to achieve performance that is both high and portable. In particular, the authors of [37] show that it is possible to use auto-tuning to improve performance of even highly-tuned algorithms. Even more relevant to our work is [38], and we agree with the authors of [38] that auto-tuning can be used as a performance portability tool, especially with OpenCL. However, auto-tuning has been used mostly to achieve performance portability between different many-core accelerators, while we also use it to adapt an algorithm to different use-case scenarios.

While there are no previous attempts at auto-tuning dedispersion for many-cores, there are a few previous GPU implementations documented in literature. The first reference in the literature is [39]. In this paper dedispersion is listed as a possible candidate for acceleration, together with other astronomy algorithms. We agree with the authors of [39] that dedispersion is a potentially good candidate for many-core acceleration, because of its inherently parallel structure, but we believe their performance analysis to be too optimistic, and their AI's estimate to be unrealistically high. In fact, we showed in this chapter that dedispersion's AI is low in all realistic scenarios, and that the algorithm is inherently memory-bound. The same authors implemented, in a follow-up paper [35], dedispersion for NVIDIA GPUs, using CUDA as their implementation framework. However, we do not completely agree with their performance results for two reasons: first,

they do not completely exploit data-reuse, and we showed how important data-reuse is for performance, and second, part of their results are not experimental, but derived from performance models.

Another GPU implementation of the dedispersion algorithm is presented in [40]. Also in this case there is no mentioning of exploiting data-reuse. In fact, some of the authors of [40] published, shortly after the first work, another short paper [36] in which they affirm that the previous algorithm does not perform well enough because it does not exploit data-reuse. Unfortunately, this paper does not provide sufficient detail on either the algorithm or on experimental details such as frequencies and time resolution, for us to repeat their experiment. Therefore, we cannot verify the claimed 50% of theoretical peak performance. However, we believe this claim to be unrealistic because dedispersion has an inherently low AI, and it cannot take advantage of fused multiply-adds, which by itself already limits the theoretical upper bound to 50%.

4.2 Background

Waves traveling through a medium may interact with it in different ways; the result of one of these interactions is called *dispersion*. The most common example of dispersion comes from the realm of visible light: rainbows. In the case of a rainbow, the original signal is dispersed when passing through raindrops and its composing frequencies are reflected at different angles. As a result, we see the components of a whole signal as if they were different ones.

The effect experienced in radio astronomy is similar. When a celestial source emits an electromagnetic wave, all the frequencies that are part of this emission start their journey together. However, because of the interaction between the wave itself and the free electrons in the interstellar medium, each of these frequencies is slowed down at a different non-linear rate: lower frequencies experience higher delays. As a result, when we receive this signal on Earth, we receive the different components at different times, even if they were emitted simultaneously by the source. Figure 4.1 shows the effect of dispersion on an impulsive radio signal. The top panel shows the arrival time versus the observing frequency, while the bottom panel shows the dedispersed pulse, which closely approximates the originally emitted signal.

More formally, the highest frequency (i.e. f_h) of a signal emitted by a specific source at time t_0 is received on Earth at time t_x, while all the other frequency components (i.e. f_i) are received at time $t_x + k$. This delay k, measured in seconds, is described by Equation 4.1; the frequency components of the equation are measured in MHz.

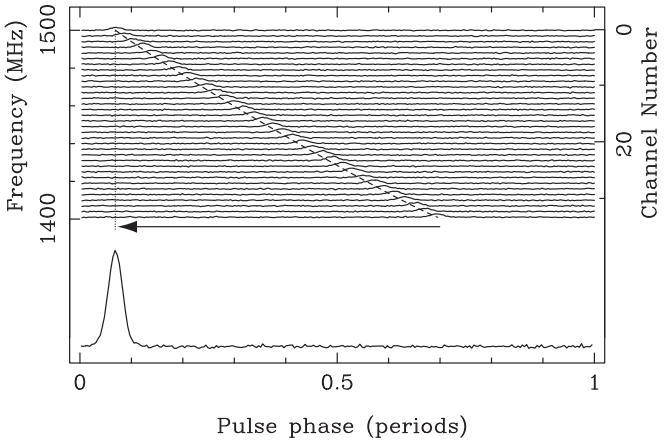

Figure 4.1: The effect of dispersion on a pulsar signal, courtesy of Lorimer and Kramer [41].

$$k \approx 4,150 \times DM \times (\frac{1}{f_i^2} - \frac{1}{f_h^2}) \tag{4.1}$$

In this equation, the *Dispersion Measure* (DM) represents the number of free electrons between the source and the receiver. Assuming a model for the distribution of these electrons along the line-of-sight, this quantity can be interpreted as a measure of the distance between the emitting object and the receiver. When observing a known object, all the quantities of Equation 4.1 are known, thus the effect of dispersion can be easily reversed. This process is called *dedispersion* and consists of shifting in time the lower frequencies in order to realign them with the corresponding higher ones, thus reconstructing the original signal.

However, applying this technique in the search for unknown astronomical objects is more difficult because DM is not known a priori. Therefore, the received signal must be dedispersed for thousands of possible DM values in a brute-force search that produces a new dedispersed signal for every trial DM. So far, no better approach is known and there are no heuristics available to prune the DM search space. The reason for this is that when the DM is only slightly off, the source signal will be smeared, and the signal strength will drop below the noise floor, becoming undetectable. It is clear from this that the process of dedispersion is computationally expensive, because every sample in the input signal must be processed for thousands of different trial DMs. In addition, modern radio telescopes can point simultaneously in different directions by forming different beams. These beams can be used to survey a bigger part of the sky and discover

astronomical sources at a faster rate. This results in even more input signals to process and greatly increases performance requirements. However, all trial DMs and beams can be processed independently, thus making it possible to improve the performance of the dedispersion algorithm by means of large-scale parallelization.

4.3 Algorithm and Implementation

Dedispersion is the process of reversing the effects of dispersion, as described in Section 4.2. We first describe the sequential dedispersion algorithm in more detail, and analyze its complexity in Section 4.3.1. We then present our parallel implementation and its optimizations in Section 4.3.2.

4.3.1 Sequential Algorithm

The input of this algorithm is a channelized time-series, i.e. a time-series with each frequency channel represented as a separate component. The time-series is represented as a $c \times t$ matrix, where c is the number of frequency channels and t is the number of time samples necessary to dedisperse one second of data at the highest trial DM. This number, t, is always a multiple of the number of samples per second. The output is a set of dedispersed time-series, one for each trial DM, and it is represented as a $d \times s$ matrix, where d is the number of trial DMs and s is the number of samples per second. During dedispersion, the frequency channels are typically integrated to reduce the data rate. Every data element in these matrices is represented using a single precision floating point number. To provide a quantitative example, the Apertif system on the Westerbork telescope will receive 36 GB/s in input, and produce 72 GB/s of dedispersed data.

The sequential pseudocode for the dedispersion algorithm is shown in Algorithm 3. Even if the algorithm looks trivially parallel, it is very data-intensive, so achieving high performance is not trivial. To simplify the discussion, and without losing generality, in this chapter we describe the case in which there is a single input beam, but all results can be applied to the case of multiple beams. The algorithm consists of three nested loops, and every output element is the sum of c samples: one for each frequency channel. Which samples are part of each sum depends on the applied delay (i.e. Δ) that, as we know from Equation 4.1, is a non-linear function of frequency and DM. These delays can be computed in advance, so they do not contribute to the algorithm's complexity. Therefore, the complexity of this algorithm is $O(d \times s \times c)$.

In the context of many-core accelerators, there is another algorithmic characteristic that is of great importance: *Arithmetic Intensity* (AI), i.e. the ratio between the performed floating point operations and the number of bytes accessed in memory. The AI is extremely important, because in many-core architectures

Algorithm 3 Pseudocode of the dedispersion algorithm.

```
for dm = 0 → d do
  for sample = 0 → s do
    dSample = 0
    for channel = 0 → c do
      dSample += input[channel][sample + Δ(channel, dm)]
    end for
    output[dm][sample] = dSample
  end for
end for
```

the gap between computational capabilities and memory bandwidth is wide, thus a high AI is a prerequisite for high performance [19]. Unfortunately, Algorithm 3 shows that dedispersion's AI is inherently low, as there is only one floating point operation for every input element loaded from global memory. A bound for the AI of dedispersion is presented in Equation 4.2, where ϵ represents the effect of accessing the delay table and writing the output.

$$AI = \frac{1}{4+\epsilon} < \frac{1}{4} \tag{4.2}$$

The low AI of Equation 4.2 identifies dedispersion as a memory-bound algorithm on most architectures, thus the performance of this algorithm is limited not by the computational capabilities of the architecture used to execute it, but by its memory bandwidth. A way to increase dedispersion's AI, thus improving performance, is to reduce the number of reads from global memory by implementing some form of data-reuse. Analysis of Algorithm 3 suggests that some data-reuse may indeed be possible. Given that the time dimension is represented with discrete samples, it may happen that, for some frequencies, the delay is the same for two close DMs, dm_i and dm_j, so that $\Delta(c, dm_i) = \Delta(c, dm_j)$. In this case, the same input element can be used to compute two different sums, thus offering the opportunity for data-reuse and an improved AI. If this data-reuse is exploited, we can compute a new upper bound for the AI; this new upper bound is presented in Equation 4.3.

$$AI < \frac{1}{4 \times (\frac{1}{d} + \frac{1}{s} + \frac{1}{c})} \tag{4.3}$$

The bound from Equation 4.3 goes towards infinity. It may be tempting to overestimate this theoretical result and believe that, by exploiting data-reuse, dedispersion's AI can be increased enough to make the algorithm itself compute-bound. However, we found this is not the case in any realistic scenario. To

approximate this upper bound, data-reuse should be possible for every combination of DMs and frequencies, but the delay function is not linear, and delays diverge rapidly at lower frequencies. This means that, in realistic scenarios, there will never be enough data-reuse to approach the upper bound of Equation 4.3. Moreover, using the same delay for every combination of DMs and frequencies would produce the same result for every dedispersed time-series, thus making the computed results useless (i.e. the DM step is too small). Therefore we conclude that, even if data-reuse is possible, it depends on parameters like DM values and frequencies that cannot be controlled and this makes the upper bound on the algorithm's AI presented in Equation 4.3 not approachable in any realistic scenario. In this conclusion we differ from previous literature like [39] and [35]. We will prove our claim with experimental results in Section 4.5.3.

4.3.2 Parallelization

The first step in parallelizing the dedispersion algorithm for many-cores is to determine how to divide the work among different threads and how to organize them; in this work we use OpenCL as our many-core implementation framework, thus we utilize OpenCL terminology when referring to threads and their organization. From the description of the sequential dedispersion algorithm and its data structures we can identify three main dimensions: DM, time and frequency. Of these three dimensions, DM and time are the ones that are independent from each other. Moreover, they also lack internal dependencies, thus every two time samples or DMs can be computed independently of each other. These properties make the two dimensions ideal candidates for parallelization, avoiding any inter- and intra-thread dependency. In our implementation, each OpenCL work-item (i.e. thread) is associated with a different (DM, time) pair and it executes the innermost loop of Algorithm 3. An OpenCL work-group (i.e. a group of threads) combines work-items that are associated with the same DM, but with different time samples.

This proposed organization has another advantage other than thread independence: it simplifies the memory access pattern making it possible to have coalesced reads and writes. In many architectures, memory accesses generated by different threads are coalesced if they refer to adjacent memory locations, so that the different small requests are combined together in one bigger operation. Coalescing memory accesses is a well-known optimization, and it is usually a performance requisite for many-core architectures, especially in case of memory-bound algorithms like dedispersion. In our implementation, consecutive work-items in the same work-group write their output element to adjacent, and aligned, memory locations, thus accessing memory in a coalesced and cache-friendly way.

Reads from global memory are also coalesced but, due to the shape of the

delay function, are not always aligned. This lack of alignment can be a problem because, on most architectures, memory is transferred in cache-lines, and unaligned accesses can require the transfer of more than a cache-line, thus introducing memory overhead. In the case of dedispersion, the delay function is part of the algorithm and cannot be modified, but, if a cache-line contains the same number of element as the number of work-items executed in parallel by a Compute Unit (CU), then the memory overhead is at most a factor two. Luckily, this property holds for most many-cores. A factor two overhead may have a big impact on performance, but this worst-case scenario applies only if the number of work-items per work-group is the same as the number of work-items executed in parallel by a CU. In the case of work-groups with more work-items, there is a high chance that the unnecessarily transferred elements will be accessed by other work-items in the near future, thus compensating the introduced overhead with a basic form of prefetching.

We affirmed, in Section 4.3, that data-reuse is possible when, for two different DMs, the delay corresponding to a particular frequency channel is the same. To exploit this data-reuse, and thus increase the algorithm's AI, the parallel algorithm needs to be slightly modified to compute more than one DM per work-group. So, the final structure of our many-core dedispersion algorithm consists of two-dimensional work-groups. In this way a work-group is associated with more than one DM, so that its work-items either collaborate to load the necessary elements from global to *local memory*, a fast memory area that is shared between the work-items of a same work-group, or rely on the cache, depending on the architecture. Therefore, when the same element is needed by more than one work-item, the accesses to memory are reduced. There is no penalty introduced with this organization, and everything discussed so far still holds because the one-dimensional configuration is just a special case of the two-dimensional one. Work-items can also be modified to compute more than one output element, thus increasing the amount of work per work-item to hide memory latency. Accumulators are kept in registers by the work-items, a further optimization to reduce accesses to global memory.

The general structure of the algorithm can be specifically instantiated by configuring four user-controlled parameters. Two parameters are used to control the number of work-items per work-group in the time and DM dimensions, regulating the amount of available parallelism. The other two parameters are used to control the number of elements a single work-item computes, also in the time and DM dimensions, regulating the amount of work per work-item. The source code implementing a specific instance of the algorithm is generated at run-time, after the configuration of these four parameters.

Platform	Cores	GFLOP/s	GB/s
AMD HD7970	2048	3788	264
Intel Xeon Phi 5110P	120	2022	320
NVIDIA GTX 680	1536	3090	192
NVIDIA K20	2496	3519	208
NVIDIA GTX Titan	2688	4500	288

Table 4.1: Characteristics of the used many-core accelerators.

4.4 Experimental Setup

In this section we describe how the experiments are carried out and all the necessary information to replicate them. We start by describing the many-core accelerators we used, the software configuration of our systems and the observational setups. We then describe the specificities of each of the three experiments that are the focus of this chapter. Table 4.1 contains a list of the many-core accelerators we used in this work, and reports some basic details for each of them. In particular, the table shows each platform's number of *Compute Elements* (CEs), peak performance and peak memory bandwidth.

We run the same code on every many-core accelerator; the source code is implemented in C++ and OpenCL. The OpenCL runtime used for the AMD HD7970 GPU is the AMD APP SDK 2.8, the runtime used for the Intel Xeon Phi is the Intel OpenCL SDK XE 2013 R3 and the runtime used for the NVIDIA GPUs is NVIDIA CUDA 5.0; the C++ compiler is version 4.4.7 of the GCC. The accelerators are installed in different nodes of the Distributed ASCI Supercomputer 4 (DAS-4). DAS-4 runs CentOS 6, with version 2.6.32 of the Linux kernel. In all experiments, the algorithm is executed ten times, and the average of these ten executions is used for the measurements. Dedispersion is always used as part of a larger pipeline, so we can safely assume that the input is already available in the accelerator memory, and the output is kept on device for further processing. There is no need in this scenario to measure data-transfers over the PCI-e bus.

The experiments are performed in two different observational setups. These setups are based on the characteristics of two radio telescopes operated by the *Netherlands Institute for Radio Astronomy* (Astron): LOFAR and the Apertif system on Westerbork. In our Apertif setup, the time resolution is 20,000 samples per second. The bandwidth is 300 MHz, divided in 1,024 frequency channels of 0.29 MHz each; the lowest frequency is 1,420 MHz, the highest frequency is 1,720 MHz. In our LOFAR setup, the time resolution is higher, 200,000 samples per second, but the bandwidth is lower, 6 MHz divided in 32 frequency channels

of 0.19 MHz each; the lowest frequency is 138 MHz, the highest frequency is 145 MHz. In both setups, the first trial DM is 0 and the increment between two successive DMs is 0.25 pc/cm^3.

The differences between the two setups are important because they highlight different aspects of the algorithm. Specifically, the Apertif setup is more computationally intensive, as it involves 20 MFLOP per DM, three times more than the LOFAR setup with just 6 MFLOP per DM. However, the frequencies in Apertif are much higher than in LOFAR, thus the delays are smaller and there is more available data-reuse. Therefore, we represent two different and complementary scenarios: one that is more computationally intensive, but offers more possible data-reuse, and one that is less computationally intensive, but precludes almost any data-reuse.

4.4.1 Auto-Tuning

Our first experiment consists of the auto-tuning of the four algorithm parameters described in Section 4.3. The goal of this experiment is to find the optimal configuration of these parameters for the five many-core accelerators, in both observational setups. Without auto-tuning we have no a priori knowledge that can guide us into selecting optimal configurations, thus the need for this experiment. Moreover, we are interested in understanding how these configurations differ, and if a generic configuration can be identified.

The algorithm is executed for every meaningful combination of the four parameters, on every accelerator, and for both operational setups. A configuration is considered meaningful if it fulfills all the constraints posed by a specific platform, setup and input instance. An input instance is defined by the number of DMs that the algorithm has to dedisperse. In this experiment we use 12 different input instances, each of them associated with a power of two between 2 and 4,096. Due to memory constraints, some platforms may not be able to compute results for all the input instances. The optimal configuration is chosen as the one that produces the highest number of single precision floating point operations per second. The output of this experiment is a set of tuples representing the optimal configuration of the algorithm's parameters; there is a tuple for every combination of platform, observational setup and input instance.

4.4.2 Impact of Auto-Tuning on Performance

Our second experiment measures the performance of the previously tuned dedispersion algorithm. The goal of this experiment is twofold. On the one hand, we want to show performance and scalability of the tuned algorithm, and evaluate the suitability of many-core accelerators to perform this task for current and

future radio telescopes. On the other hand, we want to provide a scientific evaluation of the impact that auto-tuning has on the performance of dedispersion. The byproduct of this experiment is a comparison of different platforms in the context of radio astronomy.

The algorithm is executed on every accelerator, for each combination of observational setup and input instance. For each of these runs, the optimal configuration is used (which we found in the first experiment). The metric used to express performance is the number of single precision floating point operations per second. To quantify the impact of auto-tuning on performance we present the signal-to-noise ratio of the optimums.

4.4.3 Data-reuse and Performance Limits

Our third experiment consists of measuring the performance of dedispersion in a situation in which optimal data-reuse is made possible by artificially modifying the delays. The goal of this experiment is to show how much the observational setup affects performance. Moreover, we want to provide empirical proof that, even with perfect data-reuse, it would not be possible to achieve high AI because of the limitations of real hardware, in contrast to what reported in literature.

This experiment performs the same steps of experiments 1 and 2, with the only difference that all the DM values are the same: the only DM value used is 0, thus there are no shifts. With no shifts applied, perfect data-reuse becomes theoretically possible as every dedispersed time-series is exactly the same and uses exactly the same input. We tune the algorithm in this modified setup, and then measure the performance obtained using the optimal configurations. The obtained results are compared with the ones obtained in experiment 2 to measure the impact that data-reuse has on performance, and understand the limitations that real hardware poses on performance.

4.5 Results and Discussion

In this section we present the results of the experiments described in Section 4.4. For every experiment, we first report the measured results, and then discuss them. When results are different from what one would expect, we provide a further explanation. After each experiment, we provide a short summary of the main findings. To conclude, Section 4.5.4 contains some additional performance comparisons.

4.5.1 Auto-Tuning

In this section we present the results of the auto-tuning experiment described in Section 4.4.1. We begin by examining the results for the Apertif case. Figure 4.2 shows the optimal number of work-items per work-group identified by auto-tuning. The GTX 680 requires the highest number of work-items (1,024), the Xeon Phi requires the lowest (16), and the other three platforms require between 256 and 512 work-items per work-group. The first noticeable pattern is that the optimal configuration, for each platform, is more variable with smaller input instances and then becomes more stable for bigger instances. The reason is the amount of work made available by every instance. In fact, smaller instances expose less parallelism and also have a smaller optimization space associated. When the number of DMs increases, so does the amount of work and the optimization space; thus, after a certain threshold, there are no more sensible variations in the number of work-items necessary for optimal performance.

Figure 4.3 presents the results of the experiment using the LOFAR observational setup. These results are similar to the Apertif setup, but not completely. The first difference that can be identified is that the behavior is more stable: the platforms already reach their optimum with smaller inputs. The reason is that the LOFAR setup has less available data-reuse, so it is easier to find the optimum because memory-reuse is not increased, even if there are more DMs to compute. The GTX 680 remains the platform that needs the highest number of work-items (i.e. 1,000) to achieve its top performance, and the Xeon Phi still needs the lowest. The HD7970 maintains its optimum at 256 work-items per work-group, its hardware limit for the number of work-items per work-group. Titan and the K20 can be clustered at a mid-range interval.

Overall, the two setups seem similar and the different platforms behave coherently. However, in Section 4.3 we pointed out that the total number of work-items per work-group is determined by the interaction of two parameters, with each work-group organized as a two-dimensional structure. Therefore, what looks like the same result may in fact be obtained in a complete different way. As an example, the GTX 680 seems to find the same optimum in both setups: 1,024 work-items for Apertif and 1,000 for LOFAR. However, even if the numbers are similar, they are the results of two different configurations: for Apertif the work-group is organized as a square matrix of 32×32 work-items, while for LOFAR the matrix is rectangular and composed of 250×4 work-items. Taking into account the meaning of these matrices, described in Section 4.3.2, it is possible to see that in the Apertif setup, where more data-reuse is potentially available, the auto-tuning identifies a configuration that exploits this data-reuse intensively, while in the LOFAR setup, where less reuse is available, the optimal configuration relies less on reuse and more on the device occupancy. This result is clearly important because it shows not just that the algorithm can be adapted to different

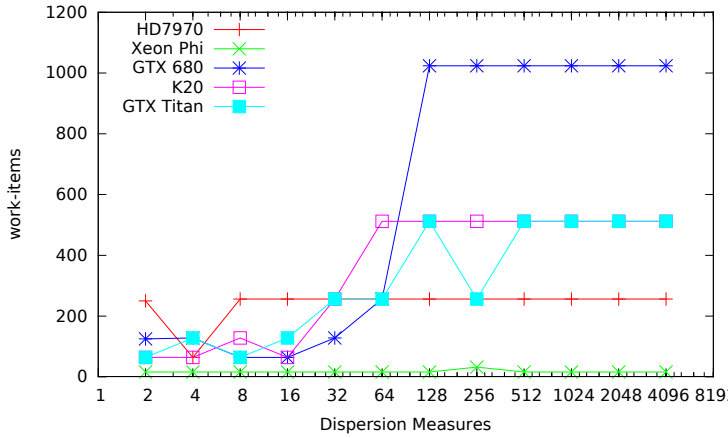

Figure 4.2: Tuning the number of work-items per work-group, Apertif.

observational setups, but also that there is no single optimal configuration.

The combination of the other two parameters of our dedispersion algorithm affects the number of registers that a work-item uses, thus affecting the amount of work that a single work-item is responsible for. Figure 4.4 illustrates the results for the Apertif setup. In the figure, the K20 and GTX Titan top the group, followed by Xeon Phi and GTX 680, while the last stand is taken by the HD7970. In these results we can spot another architectural property: K20 and Titan have the highest number of potentially available registers per work-item among our accelerators, and auto-tuning exploits this property. In fact, combining the K20 and Titan's results from both Figures 4.2 and 4.4, it is possible to see that the optimal configuration found by auto-tuning for these accelerators is to have fewer work-items than the maximum, but with more work associated.

Figure 4.5 presents the results for the LOFAR setup. The HD7970 uses the lowest number of registers, keeping its work-items lighter than the other platforms, while the Xeon Phi uses a highly variable number of them. K20 and GTX Titan use the largest number of registers, but the distance from the GTX 680 is less pronounced than in the previous setup. In this case, the optimum for these two platforms trades registers for work-items, relying more on parallelism than on heavyweight work-items. Moreover, by analyzing the composition of these results we can observe once more how auto-tuning enhances the adaptability of dedispersion. In fact, the optimal register configuration for K20 and Titan is 25×4 in the Apertif setup, and 25×2 in the LOFAR setup: also in this case, the configurations reflect the amount of possible data-reuse and show how auto-tuning makes it possible to adapt the algorithm to different scenarios.

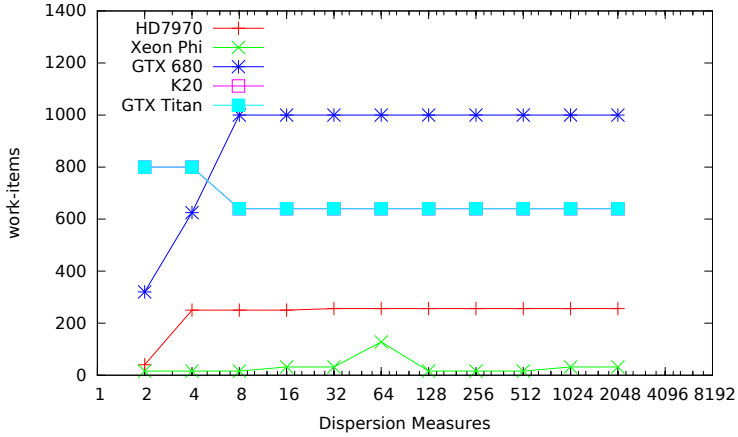

Figure 4.3: Tuning the number of work-items per work-group, LOFAR.

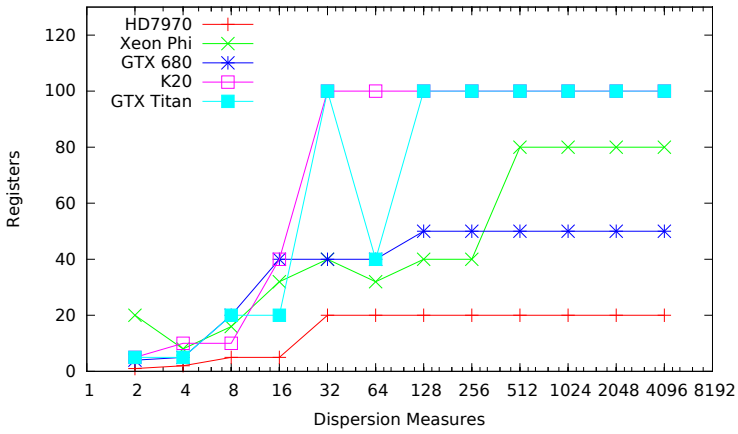

Figure 4.4: Tuning the number of registers per work-item, Apertif.

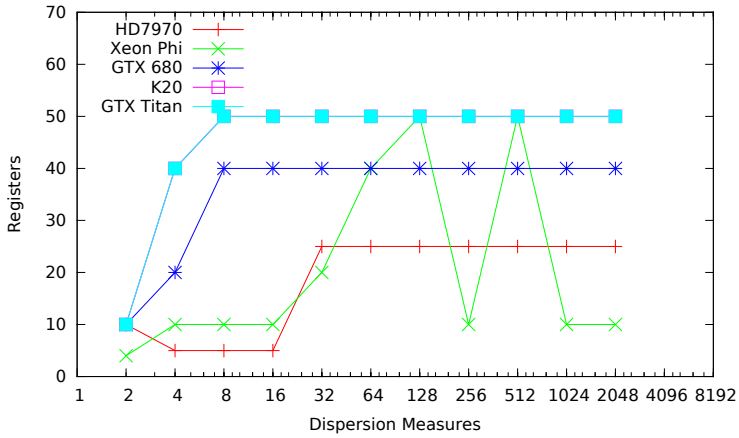

Figure 4.5: Tuning the number of registers per work-item, LOFAR.

To summarize the results of this experiment, we identified the optimal configurations of the four parameters of our dedispersion algorithm for five different many-core accelerators and two different observational setups. We identified the interactions between the configurations of these parameters, and noticed how auto-tuning provides the algorithm with the ability to adapt to different scenarios. We observed that, in general, NVIDIA GPUs require more resources, either work-items per work-group or registers, to achieve their top performance, compared with the AMD and Intel accelerators. We believe that it would not be possible to identify the optimal configurations a priori and that auto-tuning is the only feasible way to properly configure the dedispersion algorithm, because of its number of parameters, their interaction with each other, and the impact that they have on a fundamental property like the algorithm's AI. Moreover, optimal configurations are platform and observation specific, and there is no general configuration that can be used in every situation.

4.5.2 Impact of Auto-Tuning on Performance

In this section we present and analyze the results of the experiment described in Section 4.4.2. We start by introducing the results of the Apertif setup. Figure 4.6 shows the performance achieved by the auto-tuned dedispersion on the various many-core accelerators that we used in this chapter. All the platforms show a similar behavior, with performance increasing with the dimension of the input instance up to a maximum, and plateauing afterwards. The first noticeable result is that the tuned algorithm scales better than linearly up to this maximum,

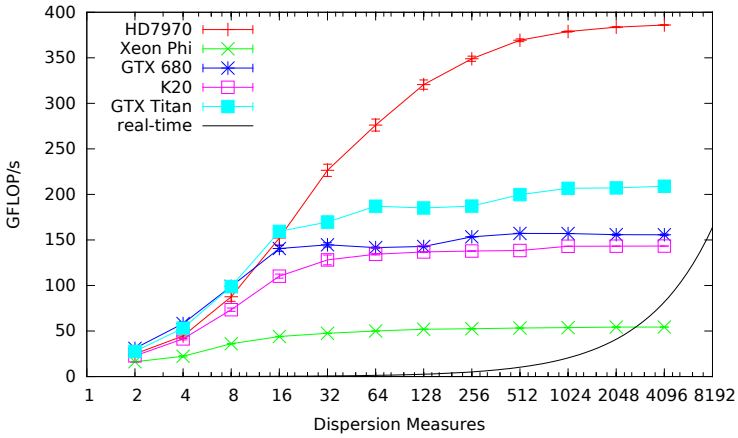

Figure 4.6: Performance of auto-tuned dedispersion, Apertif (higher is better).

and then scales linearly. The accelerators can be clustered in three groups: the HD7970 achieves the highest performance, the Xeon Phi the lowest, and the three NVIDIA GPUs, close to each other in performance, sit in the middle. On average the HD7970 is 2 times faster than the NVIDIA GPUs, and 7.5 times faster than the Xeon Phi.

The results for the LOFAR scenario are different, as can be seen from Figure 4.7. The first difference is in terms of absolute performance, with performance for LOFAR being lower than in the case of Apertif. The reason for lower performance can be found in the fact that in this setup there is less available data-reuse, thus the algorithm's AI is lower. The other difference is that the GPUs are closer to each other in performance, with the HD7970 and the GTX Titan achieving the higher performance. In fact, with less data-reuse available, the discriminant for performance is memory bandwidth, and the two GPUs with higher bandwidth achieve the top performance. In this setup the GPUs are, on average, 2.5 times faster than the Xeon Phi, but the differences between them are less pronounced than in the Apertif setup.

The line labeled "*real-time*", present in both Figures 4.6 and 4.7, represents the threshold, different for each observational setup, under which the achieved performance would not be enough to dedisperse one second of data in less than one second of computation. This constraint is fundamental for modern radio telescopes, because their extreme data rate does not allow to store the input for off-line processing. For all tested input instances, performance achieved by the auto-tuned algorithm is enough to satisfy this constraint, with the only exception represented by the Xeon Phi. Projecting these results, it can be seen that the

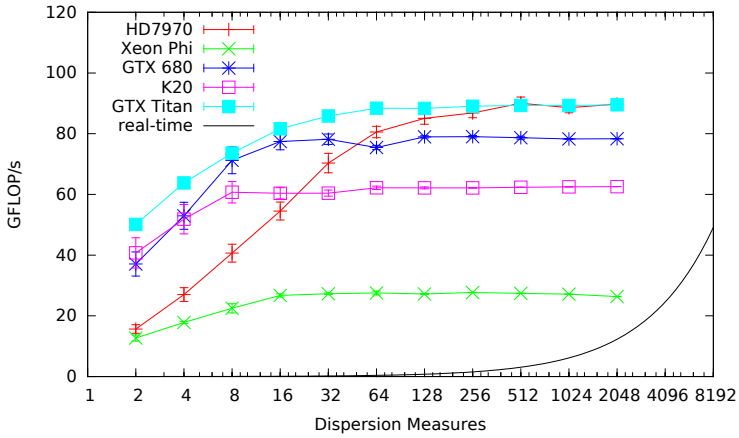

Figure 4.7: Performance of auto-tuned dedispersion, LOFAR (higher is better).

GPUs are able to scale to even bigger instances of the problem and still satisfy the real-time constraint.

This experiment aimed not just at showing performance and scalability of the tuned dedispersion algorithm, but also at measuring the impact that auto-tuning has on performance. We believe that it is important to quantify how much faster the tuned algorithm is compared with a generic configuration, and what is the statistical relevance of the optimal performance. Figures 4.8 and 4.9 present the signal-to-noise ratio of the tuned optimums, i.e. the distance from the average in terms of units of standard deviation. These results prove that, without using auto-tuning, finding the configuration that provides the best performance would be non-trivial. Applying Chebyshev's inequality [42] we can quantify an upper bound on the probability of guessing optimums with these SNRs: in the best case scenario this probability is less than 39%, while in the worst case it is less than 5%. In addition to these results, Figure 4.10 shows the shape of a typical distribution of the configurations over performance in the optimization space: it can be clearly seen that the optimum lies far from the typical configuration. In this example, there is exactly one configuration that leads to the best performance; the others perform significantly worse. We believe that the results of this experiment are an empirical proof of the importance of using auto-tuning, and a good measure of the impact that auto-tuning has on performance.

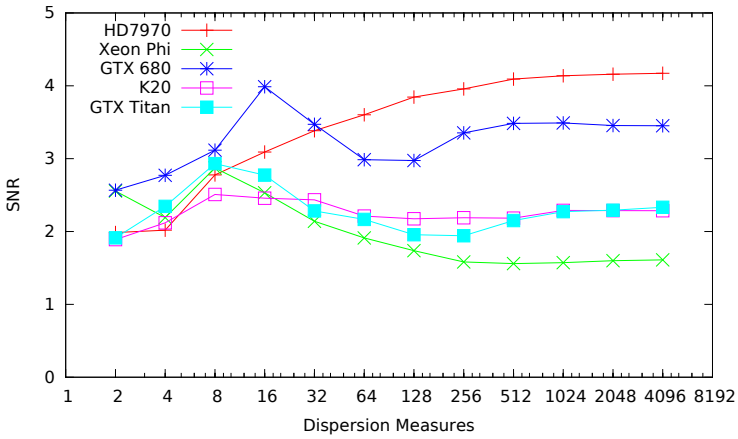

Figure 4.8: Signal-to-noise ratio of the optimum, Apertif.

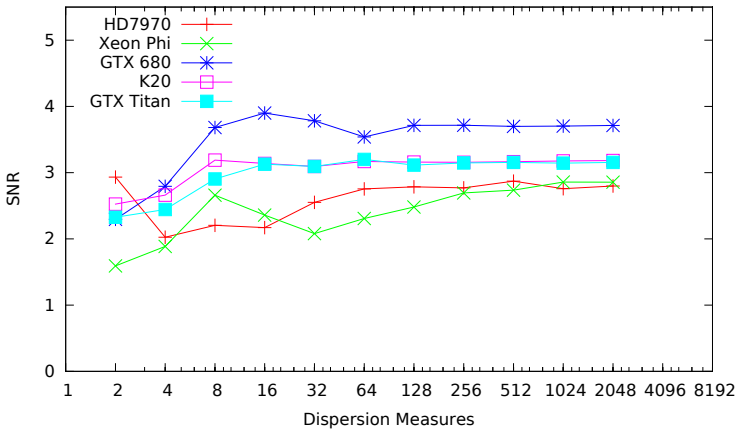

Figure 4.9: Signal-to-noise ratio of the optimum, LOFAR.

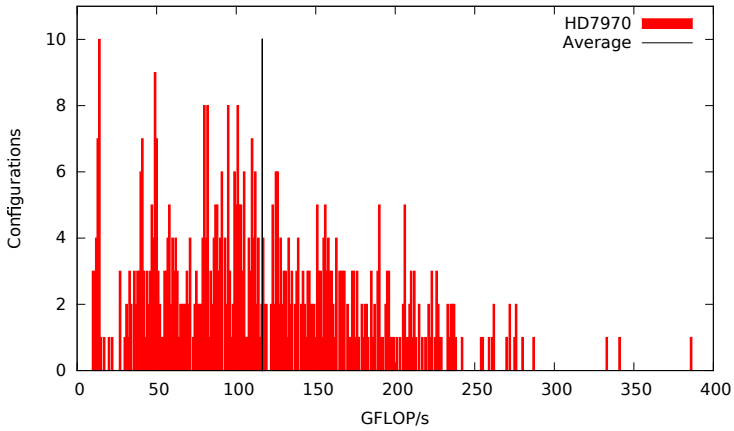

Figure 4.10: Example of performance histogram, Apertif.

4.5.3 Data-reuse and Performance Limits

In this section we present and analyze the results of the experiment described in Section 4.4.3, i.e. we tune and measure the performance of dedispersion using the same value, zero, for all DMs. Figures 4.11 and 4.12 present the results for the experiment in the Apertif and LOFAR setup, respectively. In the case of Apertif, the difference between these results and the ones in Figure 4.6 are negligible, both in terms of scalability and achieved performance. However, results for the LOFAR setup are clearly different: the performance results are higher and in line with the measurements of the Apertif setup.

The reason of this sudden change is that in this experiment both setups, even if maintaining their differences in terms of computational requirements, expose a theoretically perfect data-reuse to the algorithm, so their difference in performance is reduced. On the one hand, the fact that performance for the Apertif setup does not change between a real and a perfect setup can be explained by the fact that reuse is already maximized for the real hardware that we tested, and just exposing more data-reuse does not help increasing performance because of hardware limitations. On the other hand, performance for the LOFAR setup does change, because the increased data-reuse is exploited until the hardware is saturated.

To summarize, with this experiment we once more showed that the observational setup does affect performance. In particular, what ultimately determines performance is the amount of data-reuse that the observational setup presents to the algorithm, and this is a function of frequencies and DM values. Data-reuse

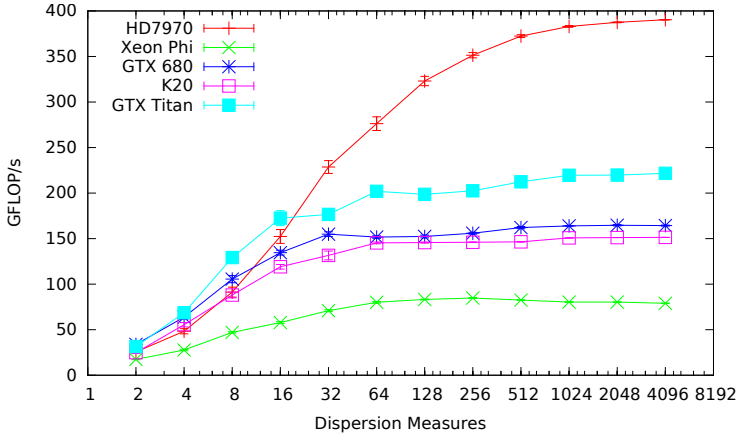

Figure 4.11: Performance in a 0 DM scenario, Apertif (higher is better).

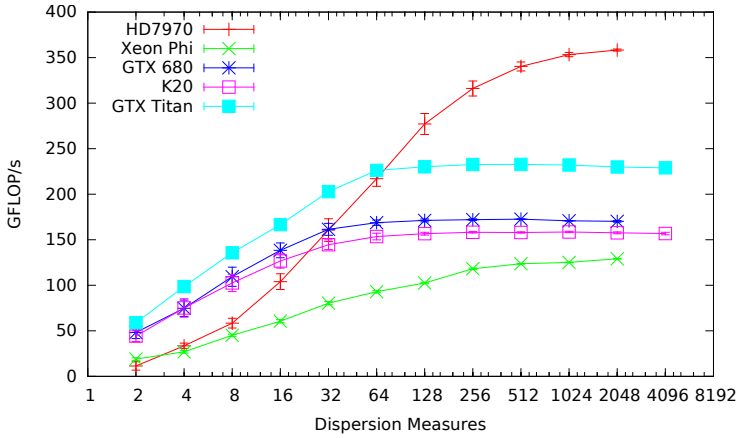

Figure 4.12: Performance in a 0 DM scenario, LOFAR (higher is better).

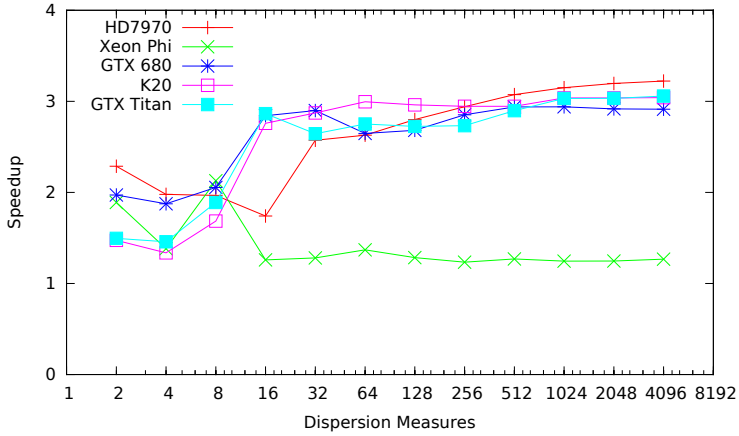

Figure 4.13: Speedup over fixed configuration, Apertif (higher is better).

is so important that, if the observational setup does not expose enough of it, the algorithm is unable to achieve its potential maximum performance. However, even when perfect data-reuse would make it possible to achieve a theoretically unbounded AI, thus making the algorithm compute-bound, limitations of real hardware do not permit to practically achieve this result. We therefore conclude that dedispersion is memory-bound for every practical and real scenario.

4.5.4 Discussion

In this section we present some results that are complementary to the previous experiments, and discuss some more general findings. We start by comparing the optimal performance of our auto-tuned algorithm to the performance of the *best possible* manually optimized version. This manually optimized version uses a "*fixed*" configuration, i.e. it uses the configuration that, working on all input instances, maximizes the sum of achieved GFLOP/s. We find the *best possible* fixed version with auto-tuning. This configuration is different for each accelerator and observational setup. Identifying a single fixed configuration that works on all accelerators and observational setups is possible, but performance would be too low to provide a fair comparison. Figure 4.13 and 4.14 show the speedup of the auto-tuned algorithm over fixed configurations.

For Apertif, we see that the tuned optimums are 3 times faster than fixed configurations for all GPUs, while the gain in performance for the Xeon Phi is less pronounced. This difference in performance between tuned and fixed configurations can be seen also in the LOFAR setup, but it is smaller than for Apertif.

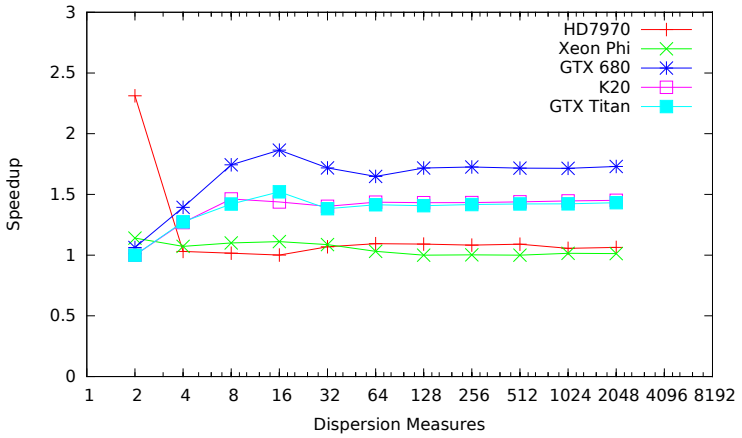

Figure 4.14: Speedup over fixed configuration, LOFAR (higher is better).

In this case, the NVIDIA GPUs still gain a 50% in performance by auto-tuning, but the HD7970 and Xeon Phi tuned configurations are only slightly faster than the fixed ones. This is because, as we noted in Section 4.5.1, the smaller optimization space of the LOFAR setup makes the optimum more stable, thus making it easier to manually tune the algorithm. We believe that these results are further evidence of the importance that auto-tuning has in achieving high performance.

We also believe that, currently and in the foreseeable future, many-core accelerators are necessary to achieve high performance for dedispersion. To provide additional strength to this claim, we compare the performance of our tuned many-core algorithm with an optimized CPU version. This CPU version of the algorithm is parallelized using OpenMP, with different threads computing different DM values and blocks of time samples. Chunks of 8 time samples are computed at once using Intel's Advanced Vector Extensions (AVX). The CPU used to execute the code is the Intel Xeon E5-2620; all the experimental parameters are the same as described in Section 4.4 except for the used compiler, which is version 13.1.1 of the Intel C++ Compiler (icc). The speedups over this CPU implementation are presented in Figures 4.15 and 4.16. These results show that our OpenCL dedispersion, running on many-core accelerators, is considerably faster than the CPU implementation in both observational setups.

With regard to the performance achieved using OpenCL on the Xeon Phi, we believe that in future a better OpenCL implementation for the Phi will certainly increase its performance, and we hope that dedispersion will be able to benefit from the high memory bandwidth of this accelerator. The aim of future investigations is to tune an OpenMP implementation of the algorithm on the Xeon

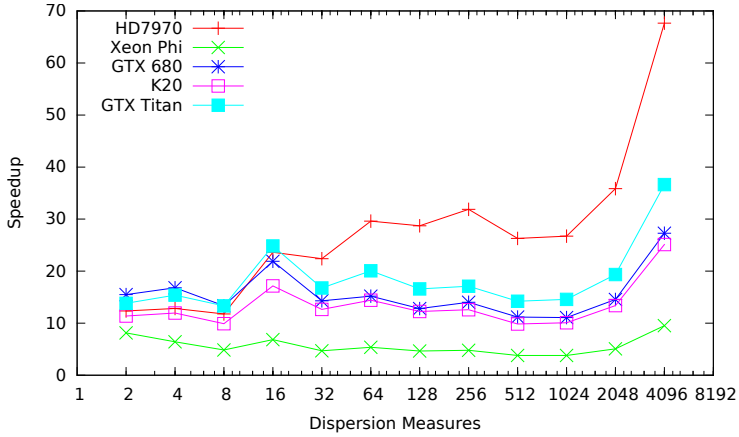

Figure 4.15: Speedup over a CPU implementation, Apertif (higher is better).

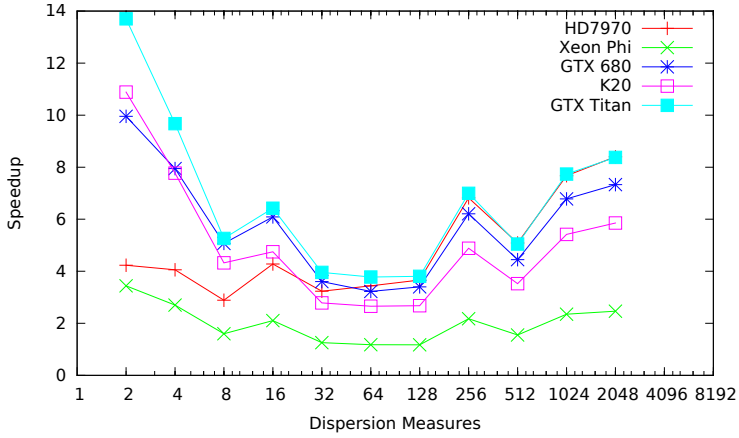

Figure 4.16: Speedup over a CPU implementation, LOFAR (higher is better).

Phi, and compare its performance with OpenCL. Furthermore, we also believe the K20 to be a poor match for a memory-bound algorithm like dedispersion, because it does not have enough memory bandwidth to feed its compute elements and keep them busy.

With the performance results of this work, we can also compute the number of accelerators that would be necessary to implement real-time dedispersion for Apertif. Apertif will need to dedisperse in real-time 2,000 DMs, and do this for 450 different beams. Using our best performing accelerator, the AMD HD7970, it is possible to dedisperse 2,000 DMs in 0.106 seconds; combining 9 beams per GPU it would still be theoretically possible to dedisperse one second of data in real-time, with enough available memory to store both the input and the dedispersed time-series. Therefore, dedispersion for Apertif could be implemented today with just 50 GPUs, instead of the 1,800 CPUs that would be necessary otherwise.

4.6 Conclusions

In this chapter, we analyzed dedispersion, a basic radio astronomy algorithm that is used to reconstruct smeared signals, especially when searching for new celestial objects. We introduced the algorithm and our many-core implementation, and analytically proved that dedispersion is a memory-bound algorithm and that, in any real case scenario, its performance is limited by low AI. With the experiments presented in this chapter, we also demonstrated that by using auto-tuning it is possible to obtain high performance for dedispersion. Even more important, we showed that auto-tuning makes the algorithm portable between different platforms and different observational setups. Furthermore, we highlighted how auto-tuning permits to automatically exploit the architectural specificities of different platforms.

Measuring the performance of the tuned algorithm, we verified that it scales linearly with the number of DMs for every tested platform and observational setup. So far, the most suitable platform to run dedispersion among the ones we tested, is a GPU from AMD, the HD7970. This GPU performs better than the other accelerators when extensive data-reuse is available, and achieves good performance also in less optimal scenarios, thanks to its high memory bandwidth. If there is less data-reuse, the GPUs that we tested achieve similar performance, but are still 2–7 times faster than the Intel Xeon Phi. Although this is partially due to the Xeon Phi's immature OpenCL implementation, we have to conclude that, at the moment, GPUs are better candidates for dedispersion. Dedispersion does thus provide further evidence that many-cores can be used to accelerate radio astronomy, providing more data to answer **RQ1** (see Section 1.1.2).

Another important contribution was the quantitative evidence of the impact that auto-tuning has on performance, contribution that will help us to answer

RQ2 (see Section 1.1.3). With our experiments, we showed that the optimal configuration is difficult to find manually and lies far from the average, having an average signal-to-noise ratio of 2–4. These results can also be used to understand how difficult is auto-tuning of many-core accelerators, the subject of **RQ4** (Section 1.1.5). Moreover, we showed that the auto-tuned algorithm is faster than manually tuned versions of the same algorithm on all platforms, and is an order of magnitude faster than an optimized CPU implementation.

Finally, our last contribution was to provide further empirical proof that dedispersion is a memory-bound algorithm, limited by low AI. In particular, we showed that achievable performance is limited by the amount of data-reuse that dedispersion can exploit, and the available data-reuse is affected by parameters like the DM space and the frequency interval. We also showed that, even in a perfect scenario where data-reuse is unrealistically high, the performance of dedispersion is limited by the constraints imposed by real hardware, and approaching the theoretical AI bound of the algorithm becomes impossible.

Chapter 5

The ARTS Transients Pipeline*

ARTS, the Apertif Radio Transient System, is a new instrument designed to survey the sky for transient radio sources, such as pulsars [41] or fast radio bursts [17]. To find new transients ARTS will process, in real-time, 444 different streams of input data received from the Apertif instrument on Westerbork [43]. Real-time processing is not just technically necessary to cope with an input data rate of more than 36 GB/s, but it is also important to allow astronomers to save a copy of the recorded data at the highest possible resolution, and to trigger follow-up observations in other frequency bandwidths or with other instruments. However, detecting radio transients in real-time, and at such high data rates, is not trivial.

In principle, detecting a transient object is a straightforward process that consists in simply monitoring a stream of data looking for a peak with high enough signal-to-noise ratio (SNR). Even considering the need to perform this process in real-time, the computational requirements associated with it would still be low. Unfortunately, electromagnetic signals generated in space interact with external factors, and what is captured on Earth looks different from the original signal. In particular, the main challenge in a radio transient pipeline is to revert the effects of dispersion. Dispersion is caused by the interaction between the emitted signal and the free electrons in the inter-stellar medium, and results in the different frequencies composing a signal traveling at different relative velocities. Therefore, what was originally a detectable peak with high SNR, becomes a scattered series of small peaks, easily drawn in the background noise. To detect the signal it is thus necessary to process the received data and reconstruct the original signal, in a process called *dedispersion*.

*This chapter has been adapted from "A real-time radio transient pipeline for ARTS" [9]

Because the effects of dispersion are bigger the farther the emitting source is from the receiver, and because in a search for new objects the distance is not known a priori, each input stream must be processed and reconstructed for a large number of possible trial distances. This brute-force search is what makes real-time transient search a difficult and interesting problem. A way to speedup this search is to process the different trial distances, and the different input streams, in parallel. Given that we already used Graphics Processing Units (GPUs) to accelerate dedispersion in Chapter 4, and that these accelerators have been used for similar, though less computationally demanding, pipelines [36] [44], in this chapter we introduce a prototype for a GPU accelerated, real-time radio transient pipeline for ARTS.

To summarize our contributions, in this chapter we: (1) introduce a new parallel and GPU accelerated radio transient pipeline, (2) show that our proposed pipeline achieves real-time performance and linear scalability, and (3) estimate the size of the ARTS system and a lower bound on its power consumption. Although our focus is on ARTS, the Square Kilometer Array (SKA) will also need a high-performance transient pipeline, one able to cope with an estimated data rate of more than 10 Pb/s [45]. Our pipeline can also be seen as a first step into addressing the challenges of the SKA.

The research question addressed in this chapter is whether many-core accelerators and auto-tuning are useful for complex radio astronomy pipelines (see Section 1.1.4). The rest of this chapter is organized as follows. First, we introduce the organization of our pipeline, and analyze its components, in Section 5.1. We continue by describing the experiment used to validate the performance of our pipeline in Section 5.2, and its results in Section 5.3. Finally, Section 5.4 summarizes our findings and conclusions.

5.1 The Radio Transient Pipeline

As highlighted in this chapter's introduction, to find new transient sources in the data captured by Apertif, ARTS will process, in real-time, the data of 444 input streams, or *beams*. Figure 5.1 contains an overview of our proposed pipeline; the whole pipeline is executed for each second of data in each of the 444 beams. Although we focus on describing our specific solution to the problem of finding radio transients in real-time, the main computational kernels of this pipeline are generic and can be used in other pipelines as well.

The main computational kernels of this pipeline are called dedispersion, SNR and thresholding. Of these three kernels, dedispersion and SNR are executed on the GPU and thresholding is executed on the host. Data are copied between host and device in two distinct memory operations; the first operation transfers the input time series on the GPU, while the second operation transfers the computed

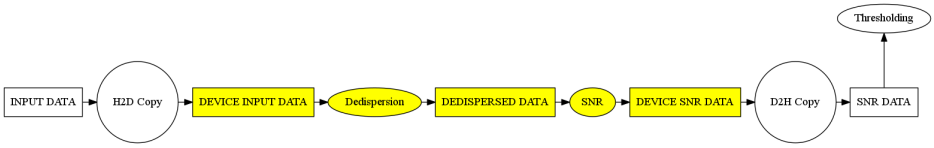

Figure 5.1: Overview of the radio transient pipeline. In the figure, ellipses represent computational kernels, circles represent memory transfers and boxes data structures. Yellow objects are stored or executed on the GPU, white objects on the host.

SNR values on the host for further processing. Intermediate data produced and consumed on the GPU are kept in device memory and not transferred back and forth.

The first, and more computationally intensive, kernel of our pipeline is dedispersion. As already mentioned in Chapter 4, dedispersion is the algorithm used to reverse the effects of dispersion. Because of dispersion, the different frequencies of the signal emitted by a radio transient source are scattered in time, causing a significant reduction in the signal's SNR; the effect of dispersion on a pulsating signal is exemplified by Figure 4.1. To reverse this effect, the different frequencies of a time series are shifted in time, relatively to each other, and realigned, before being summed together to recreate the original signal. Assuming that this shift is known, the number of operations necessary to *dedisperse* one second of data is $O(c \times s)$, where c is the number of different frequencies, called *channels*, the input data is divided into, and s is the number of samples per second. However, in the search for new transients the shift is not known in advance, thus a large number of different shifts are tried in what is known as a brute-force search; the different trial values are called *Dispersion Measures* (DMs). Therefore, in a survey with d DMs, the computational cost of dedispersing one second of data is $O(c \times s \times d)$ floating point operations per beam. If for ARTS this means a required throughput of 40 GFLOP/s per beam, and 18 TFLOP/s in total, what it means for the SKA is a required throughput of 4 TFLOP/s per beam, and 10 PFLOP/s in total. Moreover, dedispersion is a memory-bound algorithm and this makes it almost impossible to achieve peak performance on any platform. The design, parallelization and implementation details of the used dedispersion algorithm are available in Chapter 4, thus we are not going to discuss them further in this chapter.

The next computational kernel in our pipeline is called SNR. In this step, each of the d dedispersed time series generated by the previous kernel is processed, independently, to compute the SNR of the sample with highest intensity. The SNR of max, i.e. the sample with highest intensity in the time series, is defined

as $(\max - \mu)/\sigma$, where μ is the mean value of the time series and σ the standard deviation. These statistical properties of each time series are computed, in parallel and on the GPU, using Chan's algorithm [46]. In our kernel, each dedispersed time series is assigned to a different block of threads. Inside a block, each thread computes maximum, mean and standard deviation of a subset of samples, and then combines these intermediate results with the one computed by the other threads to produce the final SNR value associated with a given DM.

The last computational kernel of our pipeline is thresholding. In this kernel, the d elements array containing the SNR values is scanned to look for values that exceed a certain user specified threshold. If a value exceeds this threshold, an entry is added to the pipeline output containing the id of the DM associated with this value, and a time stamp. Before thresholding takes place, the array containing the SNR values computed by the previous kernel is copied back to the host. This is because thresholding, not benefiting from massive parallelization, is the only computational kernel of our pipeline that is not executed by an accelerator but by the host.

Our discussion so far covers the steps necessary to process a single beam, but our pipeline natively supports the processing of multiple beams. Because different beams are completely independent from each other, and can thus be independently processed, we decided to execute one instance of the pipeline for each beam. The pipeline can either be executed sequentially, one beam after the other, or in parallel, with different instances of the pipeline running at the same time, each instance associated with a different beam. Although computing different beams in parallel makes it possible to overlap the memory transfers with the execution of the computational kernels, experiments showed that this strategy does not lead to better performance on all platforms, due to driver issues. Therefore, we decided to support both modes in the pipeline, leaving the final choice to the user.

5.2 Experimental Setup

To test both performance and scalability of our pipeline, we measure its execution time on two different GPUs: an AMD HD7970 and a NVIDIA K20X. The main characteristics of these two platforms are described in Table 5.1; in particular, the table shows the number of cores that each GPU has, the theoretical peaks for single precision floating point operations per second and memory bandwidth, and the thermal design power (TDP).

The source code of our pipeline, the same for the two GPUs, is implemented in C++ and OpenCL, with OpenCL used to parallelize the computational kernels executed on the accelerators. The OpenCL runtime used for the AMD HD7970 is the AMD APP SDK 2.9, and the runtime used for the NVIDIA K20X is CUDA

Platform	Cores	GFLOP/s	GB/s	Watt
AMD HD7970	2048	3788	264	250
NVIDIA K20X	2688	3935	250	235

Table 5.1: Characteristics of the tested platforms.

Samples per Second	20000
Center Frequency	1425 MHz
Total Bandwidth	300 MHz
Channels	1024
Channel Bandwidth	292 kHz
First DM	0 pc/cm^3
DM Step	0.03 pc/cm^3

Table 5.2: Observational parameters used for the experiment.

5.5; the C++ compiler is version 4.9.0 of the GCC. The two GPUs are installed in different computing nodes of the Distributed ASCI Supercomputer 4 (DAS-4); DAS-4 uses CentOS version 6 as operating system, and version 2.6.32 of the Linux kernel.

The observational parameters used in this experiment are described in Table 5.2; they have been chosen to realistically represent one of the configurations in which ARTS will operate. The OpenCL kernels, together with the whole pipeline, were automatically tuned for both GPUs and for this specific scenario. This tuning is necessary to achieve better performance, and to provide a more fair comparison between the two devices; an in depth description of the auto-tuning process for dedispersion can be found in Chapter 4.

The pipeline is executed on each platform varying the number of trial DMs and the number of beams; for simplicity, the used values are the powers of two between 2^5 and 2^{11} for the DMs, and between 2^0 and 2^3 for the beams. Each run of the experiment involves the processing of 60 seconds of synthetically generated data; the total execution time and the execution time of each of the pipeline's steps are measured using software timers.

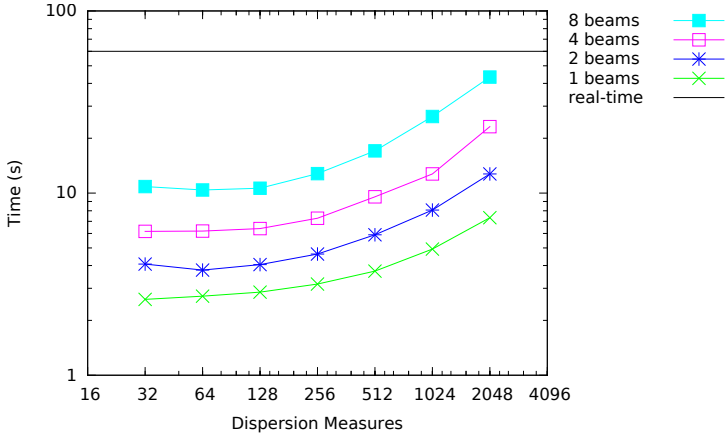

Figure 5.2: Pipeline performance on the HD7970.

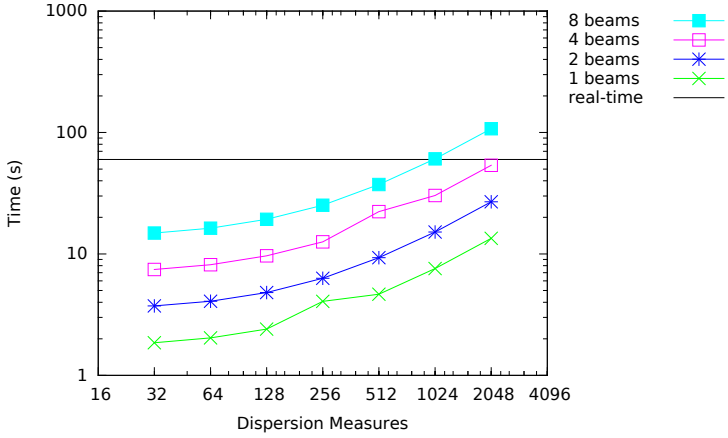

Figure 5.3: Pipeline performance on the K20X.

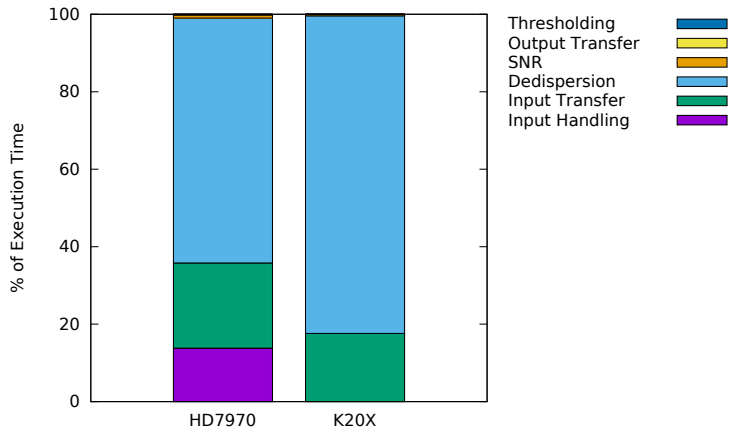

Figure 5.4: Performance breakdown of the pipeline, 1 beam and 2,048 DMs.

5.3 Performance Results

In this section we describe the results of the experiment described in Section 5.2. Figures 5.2 and 5.3 present the execution time of the pipeline, running on the HD7970 and K20X respectively; the two figures are log-log plots. The first result is that, for both GPUs, the pipeline scales linearly in both the number of beams and trial DMs. The performance, however, is different for the two devices, with the AMD GPU being faster than the NVIDIA one in almost all test cases. The reason for the difference in performance between the two devices is simple enough: the most time consuming step of the whole pipeline is dedispersion, and the HD7970 is faster at dedispersion than the K20X. While a complete analysis of dedispersion performance can be found in Chapter 4, we can say that the higher achievable memory bandwidth of the AMD GPU, coupled with its better cache system, makes it a better platform for a memory-bound kernel like dedispersion. The black line labeled "real-time" in the figures represents the threshold over which the pipeline is too slow to process the 60 seconds of input data in less than 60 seconds of execution time. As can be seen from the results, the HD7970 always satisfy the real-time requirement, while the K20X is unable to do so for a number of DMs higher than 1,024 and 8 beams.

Figure 5.4 provides a performance breakdown of the pipeline's execution time processing 2,048 DMs and a single beam. This breakdown is useful to analyze where the pipeline's bottlenecks are, and check if they are the same for each platform. From the figure we see that, for both platforms, 63 to 82 percent of the execution time is spent executing the kernels (i.e. dedispersion and SNR) on

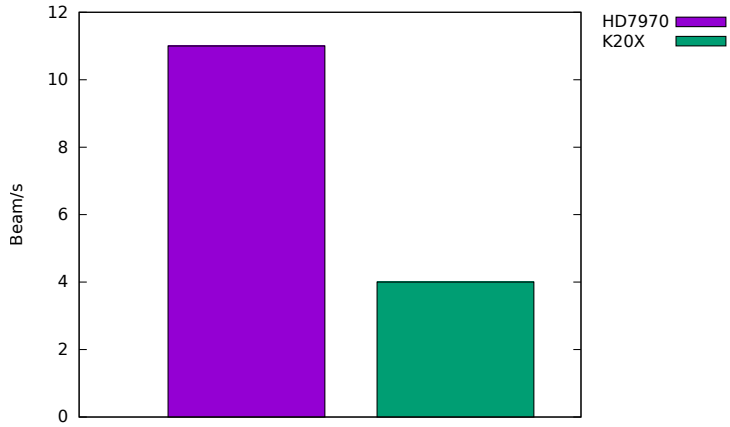

Figure 5.5: Throughput in beams per second, 2,048 DMs.

the GPU, 35 to 17 percent on memory transfers from the host to the device, and only less than 1 percent on transfering the output and thresholding. Therefore, to increase the performance of this pipeline all future efforts should be dedicated to optimizing and further improving the two computational kernels, and especially dedispersion.

Figures 5.5 and 5.6 show the throughput of our pipeline, in terms of beams and DMs that can be processed per second. The results show that the HD7970 could compute 11 beams per second, in real-time, or more than 22,000 DMs in a single beam scenario; as expected from previous results, this is more than twice the throughput achieved by the K20X. Although the results presented so far can be used to provide a comparison between these two GPUs in the context of radio transient surveys, the same results can also be used to estimate the number of GPUs that would be necessary to build the transient pipeline of ARTS.

ARTS will have to survey 444 beams in real-time, and process each of them for 2,000 different trial DMs. This means that we could have implemented ARTS in 2015, using our pipeline, with 41 AMD HD7970, or 111 NVIDIA K20X GPUs. As expected, the higher throughput of the AMD GPU makes it possible to build the same instrument with less hardware. Although it may seem exaggerated to build a system capable of hundreds of TFLOP/s for a problem that only requires tens of them, we need to stress that the compute kernels of this pipeline are all memory-bound and this makes it impossible to achieve peak performance. Still, GPUs outperform CPUs by a factor of 30 because of their higher memory bandwidth, as shown in Chapter 4.

A more compact system is not only easier to manage and less expensive to

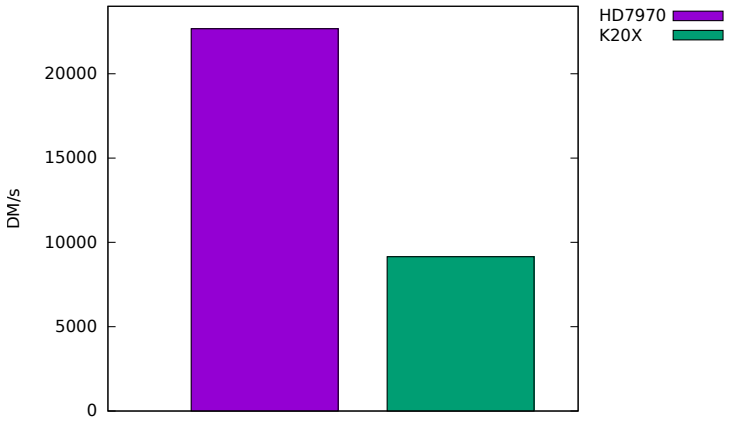

Figure 5.6: Throughput in DMs per second, 1 beam.

build, but it is also less power hungry, thus cheaper to operate. Just taking into account the TDP of the GPUs, provided in Table 5.1, we can provide here a lower bound on the power necessary to operate ARTS: 10.2 kW using the AMD solution, or 26 kW for the NVIDIA one. We believe that the introduction of new technologies, such as three-dimensional stacked memory, would help us reduce this lower bound even further.

5.4 Conclusions

In this chapter we introduced a prototype for the radio transient pipeline of ARTS. This pipeline will have to process, in real-time, 444 different input beams at a data rate of 36 GB/s, and the processing will require a throughput of more than 18 TFLOP/s. Therefore, we designed and developed a pipeline that leverages the massive amount of parallelism provided by modern GPUs. Our pipeline, implemented using C++ and OpenCL, is both high-performance and portable, and can be automatically tuned for different hardware platforms and observational scenarios.

The performance results presented in this chapter show that our pipeline scales linearly, on both tested GPUs, in the number of beams and trial DMs. Moreover, we are able to show that this pipeline can process in real-time thousands of DMs, even for more than one beam. In particular, using an AMD HD7970 GPU we show that this pipeline can process 11 beams per second at 2,048 DMs, or more than 22,000 DMs in a single beam scenario. These results

are a first step in the direction of answering **RQ3** (see Section 1.1.4), and understand if auto-tuning and many-core accelerators are useful also for complex radio astronomy pipelines, and not just single kernels. We can then conclude that, using the hardware available in 2015, ARTS could have been built using 41 GPUs, and would require 10.2 kW of power for running this pipeline.

Chapter 6

A Pulsar Searching Pipeline*

Among the most exciting challenges of modern radio astronomy is the discovery of new exotic objects like transients and pulsars. Pulsars are massive and highly magnetized neutron stars rapidly rotating on their axis, whose signal is received on Earth only periodically. Their periodicity is what makes them interesting for scientists, because it can be used to perform experiments on gravitation that are impossible to reproduce in a lab, but it is also what makes them more difficult to detect than stable sources. In fact, pulsars are so difficult to discover that since 1967, the year when the first pulsar was discovered, only around 2,400 have been found, out of a population that astronomers estimate being at least one order of magnitude bigger.

The complexity of finding new pulsars lies in the need to explore a broad three-dimensional search space, looking for a signal with a high enough signal-to-noise ratio (SNR). The three dimensions of the search space are: (1) the position in the sky, (2) the distance of the pulsar from Earth, and (3) the period. The search cannot rely on heuristics to prune the search space, and the more fine grained and extensive it is, the better are the chances of finding new pulsars. Increasing the number of points explored in this search space increases the chances of a discovery, but also boosts the computational costs of the search. Because of the high computational cost, pulsar surveys are traditionally performed off-line, that is the observational data are stored and then processed and analyzed only after the observation took place. However, the growing data rate of modern radio telescopes is pushing this strategy to the limit, making it increasingly more difficult to store the enormous amount of data produced by the instruments. This

*This chapter has been adapted from "Finding Pulsars in Real-Time" [14]

problem will only be exacerbated in the future, when telescopes like the Square Kilometer Array (SKA) will produce data rates of over 10 Pb/s [45] and require an *exascale* system for processing.

The only foreseeable solution to this problem will be to avoid collecting the data for off-line processing, and perform the search on-line and in real-time. Real-time surveys of transient astronomical sources are already reality both in the optical [47] and radio [48] domain, but the computational costs of non-periodic transient surveys are lower than for periodic sources like pulsars. Moreover, the most common technique to find periodic signals is to use the Fast Fourier Transform (FFT), but this algorithm requires the whole data set to be available for the computation, and this is not possible in a real-time scenario. Furthermore, processing the data in real-time is not the only challenge: the main challenge is to process the data in real-time and within a limited power budget. Therefore, real-time pulsar surveys are not a reality, yet.

We propose to use many-core accelerators and modern high-performance computing techniques to make finding pulsars in real-time a reality. To do so, we designed and developed a parallel pulsar searching pipeline, implemented using MPI and OpenCL. Our pipeline can run on different many-core accelerators, being them Graphics Processing Units (GPUs) or the Intel Xeon Phi, and traditional multi-core CPUs, and can be automatically tuned to adapt to a variety of telescopes and search parameters. The contributions of this chapter are: (1) demonstrating that real-time pulsar searching is possible for real instruments, (2) showing the performance, scalability and power consumption of our pipeline, and (3) comparing the use of multi-core CPUs and many-core accelerators for pulsar searching for both performance and power consumption. We believe that these contributions will remain relevant in the future because pulsar searching is a memory-bound problem, and memory bandwidth is not increasing as fast as the number of floating-point operations per second in modern hardware.

The research question addressed in this chapter is whether many-core accelerators and auto-tuning are useful for complex radio astronomy pipelines (see Section 1.1.4), and we do it by designing and parallelizing a real-time pulsar pipeline; this pipeline is more complex and challenging than the ARTS transient pipeline described in Chapter 5. The remainder of this chapter is organized as follows. Section 6.1 contains an overview of the relevant literature, while Section 6.2 introduces the basic concepts of pulsar searching, and provides a description of our real-time pipeline and all its computational kernels. The experiments, platforms and scenarios used to evaluate our pipeline are described in Section 6.3, while the experimental results are presented and analyzed in Section 6.4. Finally, Section 6.5 provides a further discussion on the results of the performed experiments, and Section 6.6 summarizes our conclusions.

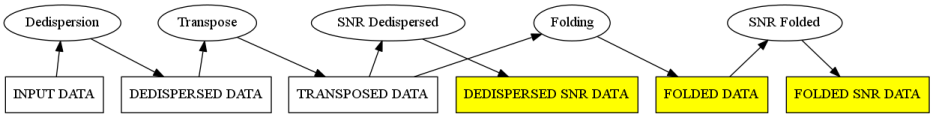

Figure 6.1: A schematic representation of the pipeline. Ellipses represent computational kernels and boxes represent data. Yellow boxes are the output.

6.1 Related Work

As mentioned in this chapter's introduction, new radio telescopes are making real-time transients search a necessity. An example is the Australian SKA Pathfinder (ASKAP) transient survey called CRAFT [48]. The pipeline described in [48] is designed to be implemented in hardware, but the paper mentions other research groups investigating the feasibility of software or FPGA based implementations; the paper also mentions the possible use of GPUs to implement the transient searching pipeline. Another real-time transient pipeline has been prototyped and tested at the BEST-2 radio telescope in Medicina [44]. This software transient searching pipeline is accelerated using GPUs, and uses the dedispersion algorithm described in [36]. Our pipeline differs from these previous works in two aspects: (1) we include a period search to also find pulsars and not only generic transient objects, and (2) we aim to design software that can be *automatically* adapted to different telescopes, instead of being specifically tailored for just one use case.

If real-time transient surveys are a reality, real-time pulsar surveys are much more difficult due to the period search that can no longer be implemented using a FFT. However, searching all the data produced by big pulsars surveys is a challenging task even when performed off-line. In 2010, the Einstein@Home (E@H) volunteer based distributed computing project has been used [49] to process previously collected data, producing the discovery of a new pulsar; the E@H distributed computing project had, at the time of publication, an aggregate computational power of 250 TFLOP/s. Additionally, even a traditional suite like DSPSR [50], a high-performance library that implements various algorithms used in pulsar astronomy, is being enhanced to exploit the computational power of many-core accelerators. Nonetheless, DSPSR does not yet support real-time pulsar searching.

6.2 Pulsar Searching

As introduced earlier in this chapter, there are three dimensions in the search space of a standard pulsars survey: (1) position, (2) distance and (3) period. To find new pulsars, all data received by the telescope needs to be processed for all

the points in this three-dimensional search space. If there is indeed a signal buried in the data, then this process will enhance its SNR and make it visible. Of these three dimensions, in the rest of this chapter we will focus only on distance and period, thus excluding the position in the sky from our discussion. The reason for excluding this dimension is that exploring it means to replicate the process described in this chapter for multiple input data streams, called *beams*. While processing multiple beams causes a linear increase in the performance requirements of the search, these beams are completely independent from each other and can thus be processed in parallel. Because of this natural parallelism, and the fact that our pipeline can effortlessly be replicated for multiple independent beams, we decided to focus this chapter on how to deal with distance and period.

An overview of our pipeline is presented in Figure 6.1; while this chapter focuses only on our parallel implementation, general information on pulsar searching techniques can be found in [51] and [41]. The first step of the pipeline is *dedispersion*; dedispersion, described in more detail in Chapter 4, is an algorithm used to reconstruct a signal assuming that it comes to our planet from a specific distance. Because this distance is one of the unknowns of the search process, the input is processed for a certain number of trial distances. In general, the higher the number of trial distances is, the higher is the chance of not leaving part of the search space unaccounted for. For *each* of these trial distances, dedispersion produces a new time series.

The following step in the pipeline is a matrix transpose that is used to rearrange the dedispersed time series in memory. The reason for this step is that the optimal memory layout for dedispersion is suboptimal for all the successive steps of our pipeline. After a series of performance experiments, we decided to add this step because the improvement in performance caused by the new memory layout by far exceeds the penalty of adding this step to the pipeline. Because matrix transposition is a well studied topic, and because this step is just an accessory part of the pipeline and not a fundamental step of pulsar searching, it will not be discussed further in this chapter. We will, however, include this step in all performance measurements.

The transposed time series are then processed by two different kernels: a SNR computation and *folding*. The SNR computation, described in more detail in Section 6.2.2, is used to analyze the dedispersed data to find non periodic sources and pulsars that are bright enough to be detected without a period search. The folding algorithm, described in more detail in Section 6.2.1, is used to perform the period search, fundamental to discover weaker pulsars. In this stage, all the dedispersed time series are folded modulo a period; like for dedispersion, this is a brute-force search and a certain number of trial periods are used. After all these stages, the input signal has been processed for a variety of trial distances and periods; a final SNR computation, described in more detail in Section 6.2.2, is

performed to analyze the folded data and identify periodic signals.

All the pipeline's stages, excluding the last SNR computation, are executed for every second of the observation. For the scope of this chapter, the input data are transferred to the many-core accelerator for every second of the computation, while the output data structures are only transferred back to the host at the end of the search; all intermediate data structures are only present on the accelerator. Because our parallel framework of choice for the accelerators is OpenCL, in this chapter we use the OpenCL nomenclature; in particular, with the word work-item we refer to a thread and with the word work-group we refer to a block of related threads executed by the same processor.

6.2.1 Folding

In off-line pulsar searching pipelines, the period search is performed using an FFT, and the input signal is folded only modulo the periods identified by the search. The reason for this is that the complexity of the FFT is lower than the complexity of folding. However, to use the FFT for the period search the whole observation must be available, and in a real-time scenario there is no option but to save at most a few seconds of data, and process them before they are replaced by new incoming data. Because folding does not require us to save the whole observation, and it can be performed for each second of data without knowledge of past or future values, we decided to use it as the algorithm for period search in our real-time pipeline. Therefore, in our pipeline we fold all the dedispersed time series modulo a number of trial periods; the number of trial periods depends on the target of the observation, as pulsar periods may range from a few milliseconds to many seconds. The pseudocode of the folding algorithm is presented in Algorithm 4; the time complexity of the algorithm is $O(d \times p \times s)$ where d is the number of trial DMs, p is the number of trial periods, and s is the number of samples per second.

Algorithm 4 Pseudocode of the folding algorithm.

```
for dm = 0 → d do
  for period = 0 → p do
    for sample = 0 → s do
      bin = computeBin(period, sample)
      output[dm][period][bin] += input[sample][dm]
      counters[dm][period][bin] += 1
    end for
    output[dm][period][bin] /= counters[dm][period][bin]
  end for
end for
```

The algorithm operates on three data structures, one input $s \times d$ matrix, and two output $d \times p \times b$ arrays, where b is the number of *bins* in which a period is divided. This algorithm can be parallelized naturally along three different, and independent, dimensions: DMs, periods and bins. In our parallel implementation, an OpenCL work-group is a three-dimensional structure, in which the first dimension is associated with the DMs, the second with the periods and the third with the bins. Each work-item in a work-group is thus naturally associated with a $(DM, period, bin)$ triple and computes the output item associated with those coordinates. This parallelization scheme maps well to both input and output data structures, so that all memory accesses performed by consecutive work-items can be coalesced [52].

6.2.2 SNR Computation

In our pipeline there are two distinct kernels computing different SNR values: the first one works on dedispersed time series, while the second one works on folded time series. The function of both steps is to separate the points in the search space where only noise is present from the points where a statistically significant signal may be present. Even if these two kernels work on different input and output data structures, they use the same formula for computing the SNR, so we can discuss them together. This formula is $(m - a)/r$, where m is the maximum value of the time series, a the mean and r the root mean square.

The first SNR computation works on the dedispersed time series, just after the transposition takes place; the input to this kernel is a $s \times d$ matrix, and the output is an array of d elements. The time complexity of this algorithm is $O(d \times s)$. The second SNR computation works on the folded time series, and differently from every other kernel of the pipeline is only executed *once* at the end of the computation. The input to this kernel is a $d \times p \times b$ three-dimensional array, and the output is a $d \times p$ matrix; the time complexity of this algorithm is $O(d \times p \times b)$. Their parallelization schemes are also similar, with the first kernel parallelized in the DM dimension, thus having each work-item associated with a particular DM, and the second kernel parallelized in both DM and period dimensions, so that each work-item is associated with a $(DM, period)$ pair.

6.2.3 Auto-Tuning

All the computational kernels described in this section expose configurable parameters to the user. By configuring these parameters the user can control various characteristics of the kernels, such as (1) the number of work-groups and work-items in a work-group, (2) the amount of work a single work-item is responsible for, (3) the amount of resources needed per work-item, (4) how programmable caches are used, (5) vectorization and (6) loop unrolling. These parameters do not

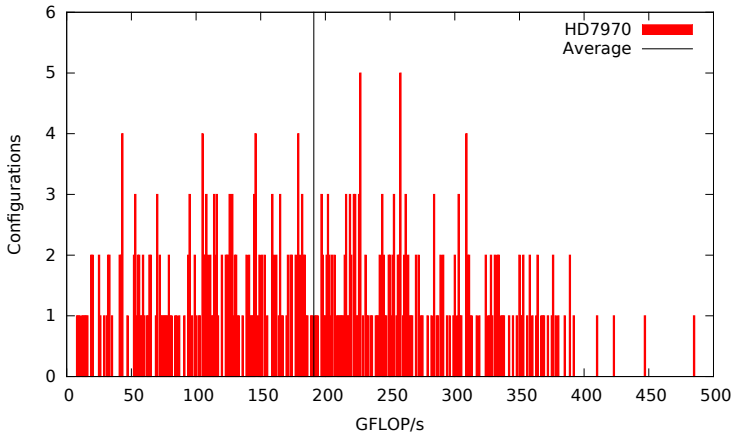

Figure 6.2: Example of auto-tuning for the dedispersion kernel.

only influence the parallelization of each kernel, but can also affect algorithmic properties like the *arithmetic intensity* (AI), i.e. the ratio between floating-point operations and bytes accessed in global memory, of a kernel. This is important because a higher AI is associated with higher attainable performance [19], especially for platforms like many-core accelerators where the gap between memory bandwidth and peak computational performance is so wide, and even more so for a memory-bound application like our pulsar searching pipeline.

However, we have no a priori knowledge about how to configure these parameters. Moreover, our goal is to execute our pulsar searching pipeline on different platforms and in different scenarios, without having to manually modify the algorithms. Therefore, we rely on auto-tuning to find the optimal configuration of each kernel's parameters, for each platform and on every scenario. To tune the pipeline we execute all kernels on each platform and for all the scenarios described in Section 6.3, measuring the achieved performance while varying all their parameters, and select the combination of kernels that achieves the higher performance. We therefore explore, for each kernel, an optimization space that can consist of hundreds of configurations, even for a single platform-scenario combination. In the performance experiments presented in this chapter we use, for each kernel of the pipeline, the optimal configuration found through the auto-tuning process.

To exemplify how much auto-tuning can affect performance, Figure 6.2 shows the performance histogram of dedispersion for one specific platform-scenario combination. What can be seen from this example is that finding the optimal configuration of a kernel without tuning is not trivial, because the optimum may lie far from the majority of configurations. Moreover, the performance obtained by

Platform	Cores	GFLOP/s	GB/s	Watt
AMD HD7970	2048	3788	264	250
NVIDIA K20	2496	3519	208	225
Intel Xeon Phi 5110P	120	2022	320	225
Intel Xeon E5-2620	6	192	42	95

Table 6.1: Characteristics of the tested platforms.

using the optimal configuration can be much higher than the performance associated with any other configuration. An in-depth analysis of the tuning of the dedispersion kernel used in this chapter can be found in [13]; the same process described in [13] has been applied in this chapter to all the other kernels.

6.3 Experimental Setup

In this section we describe all the experiments that we performed in this chapter, providing all the necessary information that can be used to replicate them. We first describe the factors common to all experiments, and then provide experiment-specific information in dedicated subsections. The many-core accelerators that we used to test our pipeline are described in Table 6.1; the table provides, for each platform, the number of OpenCL *compute elements* (CEs), the maximum dissipated power, and the theoretical peak performance and memory bandwidth. Together with the three many-core accelerators, the two graphics processing units (GPUs) and the Xeon Phi, we also included the characteristics of a multi-core Intel CPU, the Xeon E5-2620, that we will use to provide a comparison between the accelerators and a more traditional platform.

The source code[†], is the same for all tested platforms, and is implemented in C++, using MPI and OpenCL for the parallelization. The OpenCL runtime used for the AMD HD7970 GPU is the AMD APP SDK 2.9, the runtime used for the NVIDIA K20 GPU is CUDA 5.5, and the runtime used for the Intel Xeon Phi and CPU is the Intel OpenCL SDK 3.2.1; the C++ compiler is version 4.4.6 of the GCC. The accelerators are installed in computing nodes of the Distributed ASCI Supercomputer 4 (DAS-4 [‡]); the DAS-4 uses CentOS version 6 as operating system, and version 2.6.32 of the Linux kernel. All the computing nodes are connected to *power distribution units* (PDUs) that we used to monitor the power consumption of the pipeline.

[†]https://github.com/isazi/PulsarSearch
[‡]http://www.cs.vu.nl/das4/

	Apertif	LOFAR	SKA1
Samples per Second	20000	762	20000
Center Frequency	1425 MHz	142 MHz	800 MHz
Total Bandwidth	300 MHz	6.24 MHz	300 MHz
Channels	1024	8192	15000
Channel Bandwidth	292 kHz	762 Hz	20 kHz
First DM	0 pc/cm^3	0 pc/cm^3	0 pc/cm^3
DM Step	0.03 pc/cm^3	0.145 pc/cm^3	0.009 pc/cm^3
First Period	1.6 ms	41 ms	1.6 ms
Period Step	1.3 ms	1.3 ms	1.3 ms
Bins	32	32	32

Table 6.2: Characteristics of scenarios and search parameters.

The experiments are performed in three different scenarios, two based on telescopes of the *Netherlands Institute for Radio Astronomy* (ASTRON), LOFAR [53] and the Apertif [43] system on Westerbork, and one based on the baseline design for the first phase of the internationally designed Square Kilometer Array [54], SKA1. The constant parameters of these three scenarios are listed in Table 6.2. The first half of the table contains the parameters that define each scenario, while the second half contains the parameters associated with a hypothetical pulsar survey. Using different scenarios in the experiments is not only important to show the adaptability of our pipeline to different telescopes and surveys, but also because each different scenario stresses a different part of the pipeline. Moreover, by using scenarios based on the operational parameters of telescopes that are still under development, like Apertif or the SKA1, we aim at providing astronomers with insights into how to build the systems that will have to process the data of these telescopes. The variable component of each experiment will be the number of DMs and periods used for the search. Both DMs and periods will vary between the seven powers of 2 that range from 32 to 2,048; thus, the total number of tested instances per experiment will be 49 for each platform-scenario combination.

6.3.1 Pipeline Scalability

The first experiment consists in measuring the performance of our pulsar searching pipeline. The goal of this experiment is to test the feasibility of real-time pulsar searching, and show the scalability of our pipeline in terms of the number of DMs and periods. The pipeline is executed on every platform, for each sce-

nario and every combination of DMs and periods; each of the pipeline's kernels use the optimal configuration, found through auto-tuning, for the specific test case. The metric used to measure performance is the execution time, measured in seconds and using software clocks, to complete the processing of a single beam of 60 seconds.

6.3.2 Power Consumption

The second experiment consists in measuring the power consumed by our pulsar pipeline during its execution. The goal of this experiment is to identify the more power efficient platform for real-time pulsar searching. The pipeline is executed on every platform, for each scenario and every combination of DMs and periods; the same optimal configurations used for the experiment described in Section 6.3.1 are used. Because the granularity of the PDUs is a whole compute node, the measured values are normalized using the power consumption of the same node in an idle state; in this way we can take into account just the power consumed by the pipeline. The metric used in this experiment is the total consumed power, measured in kW using the DAS-4 PDUs, used by the pipeline to process a single beam of 60 seconds.

6.4 Results

In this section, we present the results of the experiments described in Section 6.3. For each experiment we first introduce the results, and then provide an analysis of them. We conclude each experiment with a summary of the findings.

6.4.1 Pipeline Scalability

The first experiment, described in Section 6.3.1, aims at measuring the performance of the pipeline to understand the feasibility of real-time pulsar searching and the scalability of the pipeline in terms of DMs and periods. All figures in this section are log-log plots and each line represents a different number of DMs; a black line, labeled "Real-Time", represents the threshold over which the execution time of the pipeline exceeds the observation time (i.e. 60 seconds) thus breaking the real-time constraint. We start presenting the results obtained with the first platform, the AMD HD7970 GPU.

Figure 6.3 presents the results of the Apertif setup. In this scenario, the pipeline achieves real-time performance in all but one of the test cases. The pipeline scales better than linearly for all DMs until the threshold of 256 periods, because there are still resources available for the computation; after this point, however, the resources available to the GPU are saturated and the pipeline scales

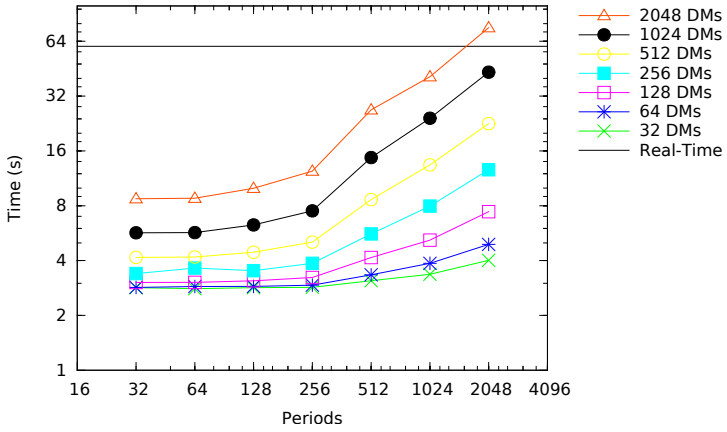

Figure 6.3: Pipeline performance for the AMD HD7970, Apertif scenario.

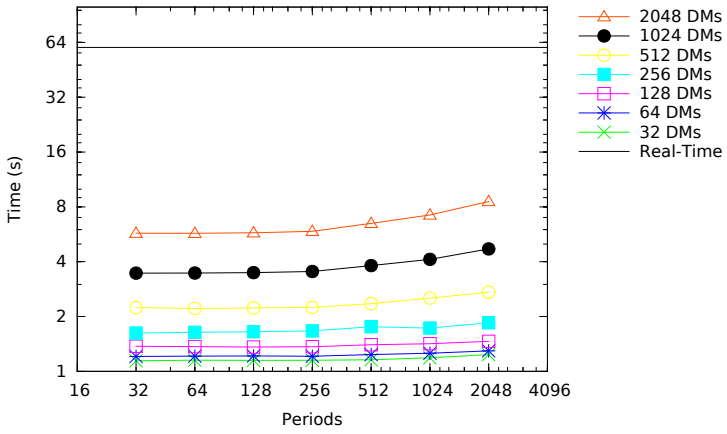

Figure 6.4: Pipeline performance for the AMD HD7970, LOFAR scenario.

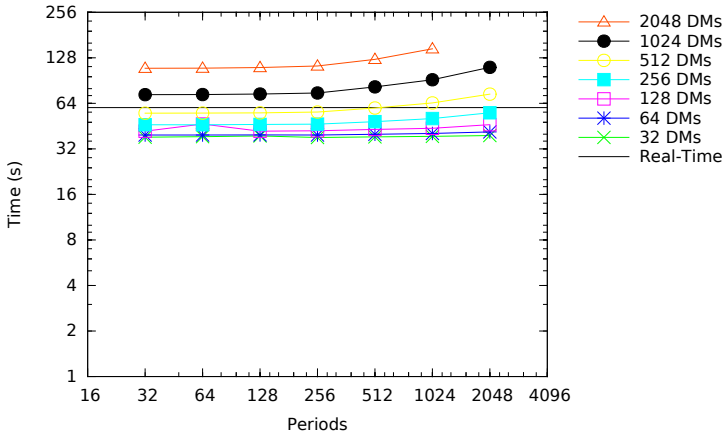

Figure 6.5: Pipeline performance for the AMD HD7970, SKA1 scenario.

linearly in the number of periods. While increasing the number of periods affects the pipeline's scalability, increasing the number of DMs has a lower impact on performance. As a matter of fact, the pipeline scales better than linearly in the number of DMs for most of the test cases, approaching linear scalability only for the bigger ones.

In the LOFAR scenario, presented in Figure 6.4, performance requirements are lower, and the pipeline satisfies the real-time constraint in all test cases. In fact, even the biggest of test cases, the $2,048 \times 2,048$ case, is 7 times faster than real-time. Achieving faster than real-time performance means that the pipeline can scale to even bigger search spaces, or process the data streams associated with multiple beams. Therefore, it would be possible to process 7 beams per GPU and still satisfy the real-time constraint even for the biggest of test cases. Furthermore, the pipeline scales better than linearly in both the number of DMs and periods.

Figure 6.5 introduces the results for the most computationally challenging scenario, SKA1; one of the data points, the one associated with the $2,048 \times 2,048$ case, is missing due to the platform's memory limitations. In this scenario, most of the test cases satisfy the real-time constraint, but only up to 512 DMs. After this point, all results lie over the real-time threshold; in correspondence with the bigger computed test case, the pipeline is 2.4 times slower than real-time. If in the LOFAR scenario better than real-time results indicated that the pipeline could scale to even bigger search spaces, pulsar surveys in the SKA1 scenario would have to be distributed: this can be done by replicating the data streams to multiple accelerators, and executing the pipeline on subsets of the global search

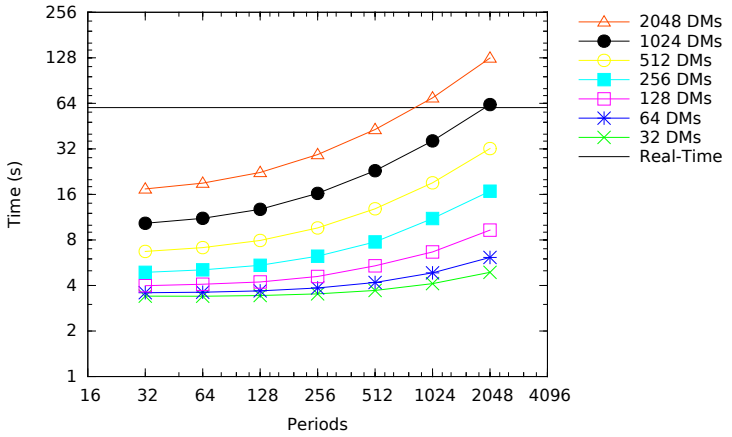

Figure 6.6: Pipeline performance for the NVIDIA K20, Apertif scenario.

space. However, better than linear scalability is achieved in both the number of DMs and periods also for the SKA1 scenario.

The second platform we tested is the NVIDIA K20 GPU; Figure 6.6 presents the results of the Apertif scenario. Like for the previous platform, we notice that almost all test cases satisfy the real-time constraint. The pipeline's execution time is higher on the K20 than on the HD7970, but the scalability is better. In fact, while scalability in both dimensions is better or close to linear, the figure shows that this platform scales more smoothly than the previous one in the number of periods.

While in the Apertif scenario we can identify differences in how the two GPUs scale, Figure 6.7 shows that results in the LOFAR scenario are almost identical. The real-time constraint is satisfied for all test cases, and in the biggest one the pipeline is 4.7 times faster than real-time. The pipeline scales better than linearly in the number of periods, and linearly or better in the number of DMs.

The SKA1 scenario is presented in Figure 6.8, and the first noticeable result is that most of the test cases do not satisfy the real-time constraint. In fact, starting from 256 DMs, no line lies below the real-time threshold. The presence of some lines below the threshold suggests that SKA1 pulsar surveys will have to be distributed among multiple K20 GPUs in the DM dimension. Although it is not always possible for the K20 to satisfy the real-time constraint in this scenario, the achieved scalability is better than linear in both the number of DMs and periods.

The last accelerator we tested is the Intel Xeon Phi. Figure 6.9 shows the performance results for this accelerator in the Apertif scenario. Most of the test

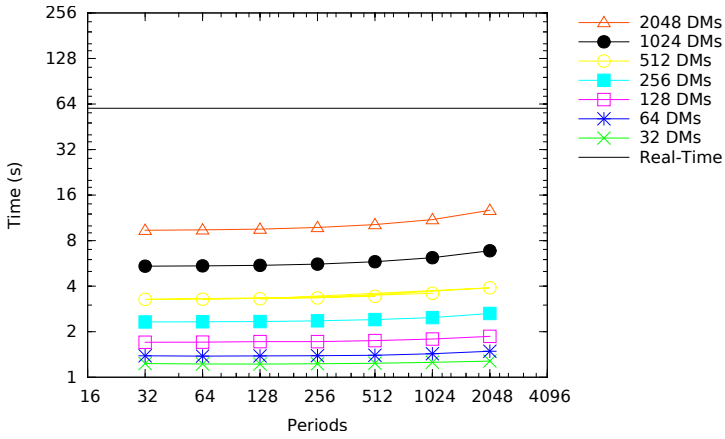

Figure 6.7: Pipeline performance for the NVIDIA K20, LOFAR scenario.

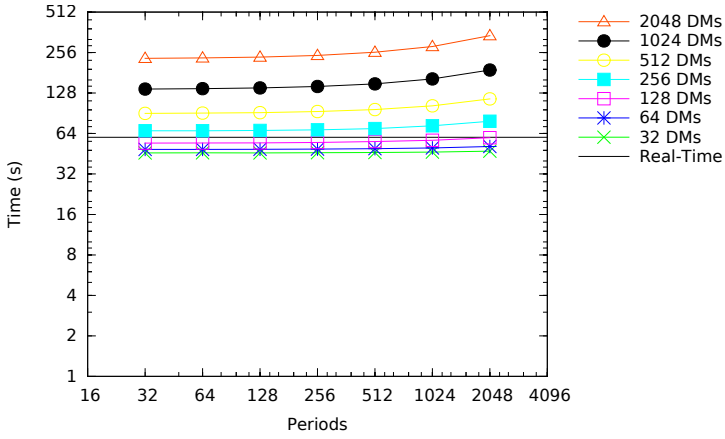

Figure 6.8: Pipeline performance for the NVIDIA K20, SKA1 scenario.

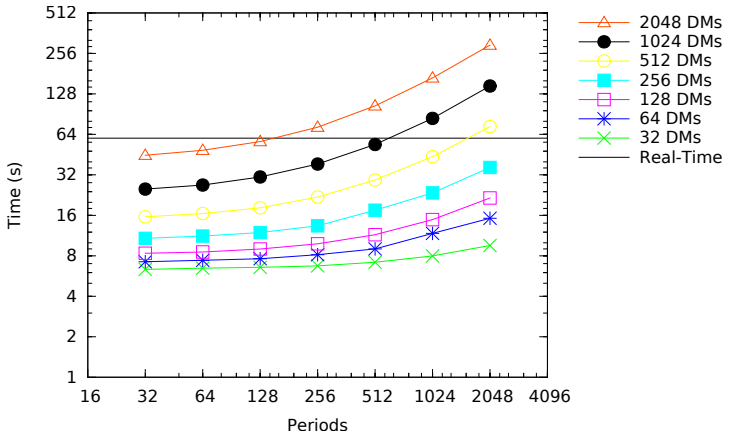

Figure 6.9: Pipeline performance for the Intel Xeon Phi, Apertif scenario.

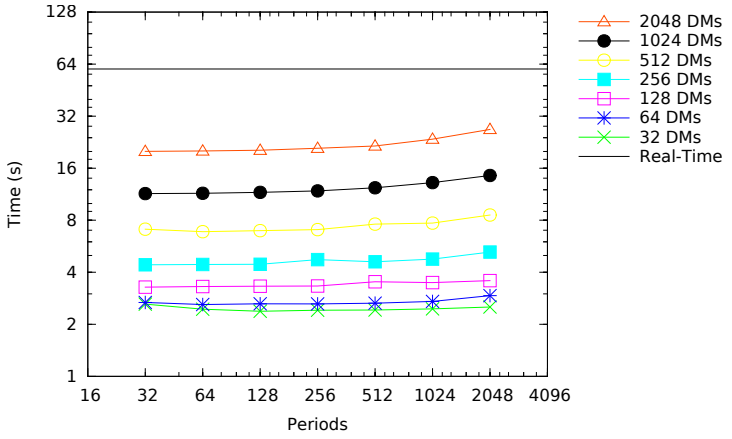

Figure 6.10: Pipeline performance for the Intel Xeon Phi, LOFAR scenario.

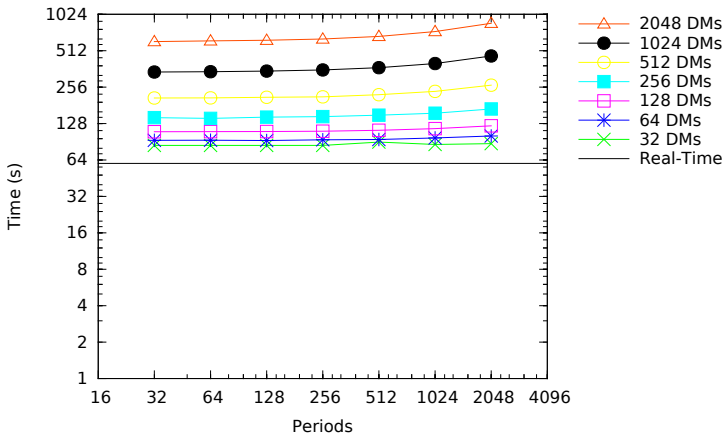

Figure 6.11: Pipeline performance for the Intel Xeon Phi, SKA1 scenario.

cases still lie below the real-time threshold, but the number of instances that do not satisfy this requirement is higher than for the GPUs, with the biggest test cases executing 4.9 times slower than real-time. The execution time, compared with the GPUs, is also higher for all test cases. The pipeline's scalability is better than or equal to linear, and the observed trend is similar to the one of the NVIDIA GPU.

The results for the LOFAR scenario are presented in Figure 6.10. In this scenario, as seen for the other platforms, all test cases satisfy the real-time constraint, even if performance is lower on the Phi than on the GPUs; the biggest test case is 2.2 times faster than real-time. Scalability in both the number of DMs and periods is better than linear.

Results for the last scenario, SKA1, are presented in Figure 6.11. Performance obtained in this scenario by the Xeon Phi is too low to satisfy the real-time constraint in any of the test cases, with the biggest one resulting 14 times slower than real-time, and the smallest one still 1.4 times slower. Achieving real-time performance in this scenario, even if theoretically possible by computing different DMs on different Xeon Phis, would require too many devices to be feasible in practice. Like for the other platforms, scalability in both the number of DMs and periods is better than linear in this scenario.

To summarize the results of this experiment, all platforms achieve linear or better than linear scalability in all scenarios in both the number of DMs and periods. The performance achieved by each platform is, however, different, with the HD7970 GPU resulting the fastest among the platforms we tested; both GPUs performed better than the Xeon Phi in all scenarios. Although the scenarios are

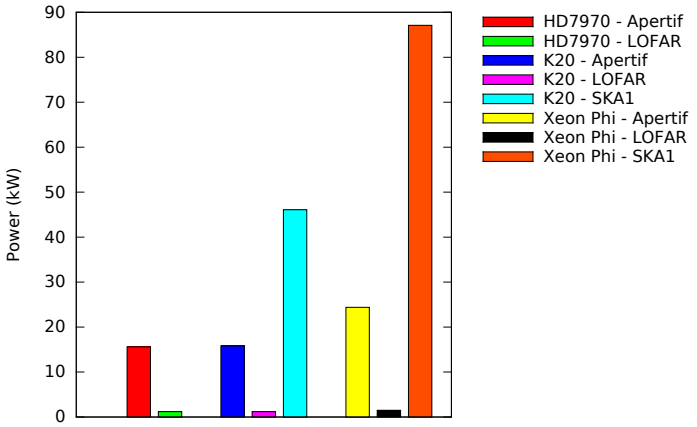

Figure 6.12: Total consumed power, $2,048 \times 2,048$ case.

different in their performance requirements, the results of this experiment show that our pipeline would be able to satisfy the real-time requirement for all of them by distributing the pipeline over multiple many-core accelerators where performance on a single accelerator would not be enough. Therefore, we can conclude that real-time pulsar searching could already be a reality for current radio telescopes, and it will enable future telescopes to perform bigger and more accurate surveys.

6.4.2 Power Consumption

In this section we present the results of the experiment described in Section 6.3.2. Due to space limitations, we will not present the results for all the test cases like we did in Section 6.4.1, but use a specific test case to highlight the results. The test case we selected is the biggest one, with 2,048 DMs and periods. We believe that it makes sense to use this particular test case because we know, from the results previously presented, that it is the test case with the higher execution time in each of the scenarios. Therefore, even if not representative of every other test case, it still provides an upper bound on the power consumed by the pipeline running on our three many-core platforms.

The results are presented in Figure 6.12; because the AMD HD7970 was not able to execute this test case in the SKA1 scenario due to memory limitations, the value is missing from the figure. However, the almost identical results obtained by the two GPUs in the Apertif and LOFAR scenarios let us believe that the total power consumption would be similar also for the SKA1 scenario. The

two GPUs require 15 and 1.2 kW to execute the pipeline for the Apertif and LOFAR scenario, respectively. Because we know from Section 6.4.1 that the execution time of the two platforms is different, we can conclude that the energy per second consumed by the K20 is lower than the energy per second consumed by the HD7970. It is also clear, from these results, that the more computationally expensive the scenario is, the more power is required for the computation, which is what we expect.

The Xeon Phi consumes more total power executing the pipeline, but this is caused only by the higher execution times and not by a higher amount of energy consumed per second. In fact, the Xeon Phi is the accelerator that, per second, consumes less energy to execute the pulsar searching pipeline. Comparing the power consumption between the GPUs and the Phi, the Phi uses only between 1.2 and 1.8 more power to complete the execution of the pipeline, while being on average 2.5 times slower than the GPUs.

To summarize the results of this experiment, the power required during the execution of the pipeline is, in all cases, sensibly lower than the thermal design power of the accelerators. Among the GPUs, the K20 is less power hungry than the HD7970, but because of the higher execution time requires the same total power as the AMD GPU to execute the pipeline. The accelerator that consumes less energy per second is the Xeon Phi, however the performance gap between it and the GPUs is too wide and this platform ends up consuming more power, in total, than the other two accelerators.

6.5 Discussion

In this section we provide further analysis of the results presented in Section 6.4, and introduce additional data to complement the experiments already presented. We start by analyzing the differences between the three scenarios that we used for our experiments. Figure 6.13 provides a breakdown of the percentage of time spent on each of the pipeline stages during the execution; for simplicity, and due to space limitations, we only show the biggest test case. The first result is that, of all of the pipeline stages, only three determine the total execution time: input handling, dedispersion and folding. All the other kernels (i.e. the transpose and the two SNR computations) combined, account only for few percentage points of the total execution time. Therefore, any future performance improvement of this pipeline will have to focus on the three main stages.

Another interesting result is that the relative impact of the three main components of the pipeline on the total execution time differs by scenario, but not so much by platform. In fact, we can see the same pattern for each platform: in the Apertif scenario folding dominates performance, while handling and transferring the input to the accelerator determines performance for the LOFAR scenario,

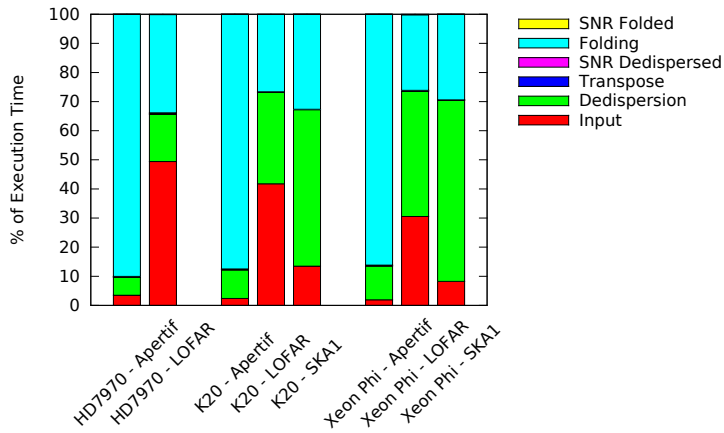

Figure 6.13: Performance breakdown, $2,048 \times 2,048$ case.

and dedispersion is the main contributor to the SKA1 execution time. These different scenarios were chosen because, for their characteristics, they would stress different aspects of the pipeline, however they are modeled on the operational parameters of real telescopes, showing that different instruments require focus on different stages of the same pipeline. In this context, we believe that having a pipeline where each kernel can be tuned and optimized *automatically* for each particular scenario is extremely important to perform real-time pulsar surveys.

Figure 6.13 provides also an explanation for the scalability trends observed in Section 6.4.1. In the Apertif scenario, performance results for all three platforms showed that the pipeline scaled better than or close to linearly in the number of DMs, but only linearly in the number of periods, and this is because folding dominates performance in this scenario. Similarly, the pipeline scales better than linearly in the number of periods for both LOFAR and SKA1 because folding is not the main performance driver in these scenarios. Because input handling dominates the LOFAR scenario, the pipeline scales better than linearly in both dimensions, but will scale linearly with an even bigger DM space. As for the SKA1 scenario, it will also approach linear scalability with more DMs to search, as dedispersion is the main factor affecting the pipeline's performance in this scenario.

Figure 6.14 provides the speedup achieved by our pipeline on many-core accelerators, compared with the same pipeline tuned and running on a multi-core CPU, the Intel Xeon E5-2620; the characteristic of this platform are also presented in Table 6.1. Also for this experiment, only the biggest test case is presented. The achieved speedup ranges between 1.2 and 8.3, with the highest speedup achieved

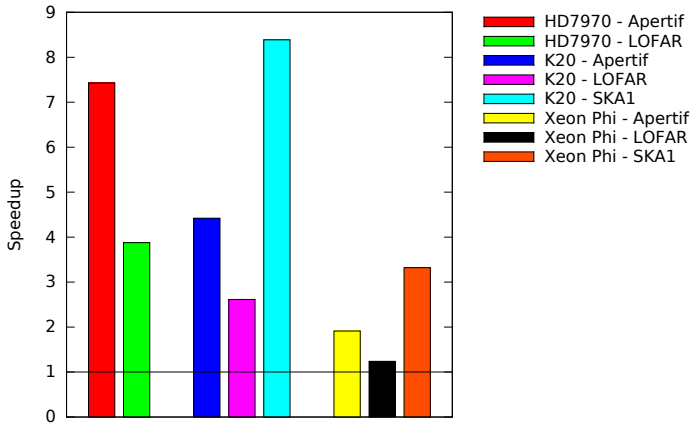

Figure 6.14: Speedup over CPU, $2,048 \times 2,048$ case.

in the SKA1 scenario, and the lowest one in the LOFAR scenario. This result is useful to highlight how many-core accelerators contribute to achieve real-time performance in pulsar searching, especially in the context of future telescopes like the SKA. Achieving high-performance means also that less hardware is necessary to process the same amount of data, lowering the costs associated with maintaining and operating the systems used for this task. Even if higher speedups are possible for individual kernels (e.g. for dedispersion [13]), obtaining higher speedups is more difficult for the pipeline as a whole as performance is always determined by the slowest kernel.

By analyzing Figure 6.15 it can be seen that many-core accelerators also require less power than using CPU. Comparing Figure 6.14 and 6.15 we see that the GPUs reduce their advantage over the multi-core CPU, as their energy consumption is much higher, although they are still 1.8–5.9 more power efficient. While the GPUs reduce their gain over the CPU when power consumption is taken into account, the Xeon Phi maintains its advantage, and achieves the same power efficiency and speedup when compared to the tested CPU. This is because the measured energy consumed per second while executing the platform is almost the same for the two Intel devices, but the execution time is lower for the Phi. Overall, many-core accelerators are a viable platform for the execution of a real-time pulsar searching pipeline, both from performance and energy efficiency perspectives.

These results can also be used to estimate the number of accelerators that we would have needed in 2015 to build a pulsar searching system for the first phase of the SKA. From the SKA1 baseline design [54] we know that, in the operational

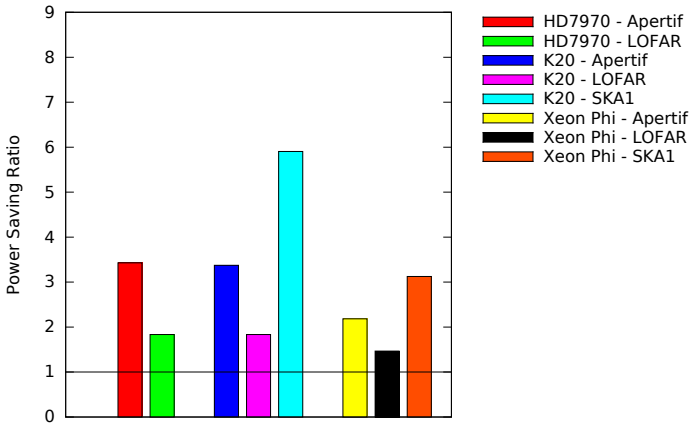

Figure 6.15: Power saving ratio over CPU, $2,048 \times 2,048$ case.

mode that we used for our experiments, a pulsar survey will explore a search space of 2,222 beams and 16,113 DMs; using the best performing accelerator, the AMD HD7970, we would need 63 GPUs per beam to dedisperse all the DMs and search to up to 2,048 periods. Therefore, the total number of GPUs to process all beams would be nearly 140,000, and this would require more than 30 MW of power just for pulsar searching. However, the number of GPUs necessary could be reduced down to only 20 per beam by having the input already available in the accelerator memory, without having to transfer it just for this pipeline. In this case, the total number of GPUs would be reduced to less than 44,500 and the power required for the search to 9.5 MW. Future improvements in the power efficiency of many-core accelerators will help reduce even more the power budget for the SKA.

6.6 Conclusions

In this chapter we introduced a new pulsar searching pipeline. All the stages of this pipeline are parallel, implemented using OpenCL, and can run on a variety of many-core accelerators. We use OpenCL to achieve code portability between different platforms, but we also provide adaptability to different telescopes and search parameters. To adapt the code, we exploit auto-tuning, finding the optimal configuration for each of the pipeline kernels in every specific scenario. The pipeline can also be distributed over multiple nodes using MPI, thus further increasing its scalability.

The main contribution of this chapter is to show the feasibility of a real-time pulsar searching pipeline, as new radio telescopes will force a shift from the traditional off-line processing to a more challenging real-time scenario, and we showed in this chapter that this is possible even using currently available hardware. We showed in Section 6.4.1 that, in different scenarios and using different platforms for its execution, our pipeline is able to process data in real-time, i.e. to process one second of input data in less than one second. In cases where we cannot show real-time performance using a single accelerator, we have the possibility to distribute the search over multiple nodes, thus satisfying the real-time constraint.

The pulsar pipeline introduced in this chapter does not only achieve real-time performance, but shows linear or better than linear scalability in all search dimensions: number of DMs and periods. This is another important result, because it allows astronomers to easily predict the number of accelerators needed for a specific search, and to minimize the number of necessary accelerators. This allows for smaller and less expensive systems, and it helps keeping under control the power required to operate them.

In Section 6.5 we showed that using many-core accelerators provides good speedup over multi-core CPUs, with GPUs being 2–8 times faster than an Intel Xeon CPU. Not only are many-core accelerators faster than a traditional CPU in executing our pulsar searching pipeline, but they also consume 1.8–5.9 less power in our tested scenarios, another characteristic that makes these platforms the most suitable for pulsar searching in the SKA era. Overall, we can affirm that combining many-core accelerators and auto-tuning is useful to improve performance, and performance portability, of this pulsar searching pipeline. Combined with the results presented in Chapter 5, this chapter allows us to affirmatively answer **RQ3** (see Section 1.1.4).

Chapter 7

Difficulty of Auto-Tuning

Auto-tuning is a well known optimization technique in computer science. So far, research has mainly focused on two aspects: (1) measuring the performance improvement that can be gained with auto-tuning compared to some baseline implementation, and (2) reducing the size of the optimization space, therefore speeding up the tuning process. In contrast, our research concentrates on measuring the importance of auto-tuning itself, and quantitatively analyzing the optimization space. For instance, we investigate how large the performance differences are between the global optimum and all other valid points in the optimization space. This way, we can make quantitative statements about the difficulty of auto-tuning different applications on different platforms. To be more precise, we define a novel generic methodology to quantitatively analyze (the impact of) auto-tuning, introducing concepts like *tuning difficulty* and *optimum portability*, and apply this methodology to many-core accelerators. With many-core accelerators becoming an integral part of supercomputers all around the world, we believe that understanding the impact that auto-tuning has on their performance is a relevant matter for the HPC community.

Studying the tuning difficulty of different algorithms, or even different classes of algorithms, on many-core accelerators is not only important for understanding auto-tuning itself, but it is also complementary to the research on reducing the size of the optimization space of a particular algorithm. The easier auto-tuning of a particular algorithm is, the closer its optimal configuration will be, in terms of performance, to all other valid configurations. Therefore, for a minimal penalty in performance, it is possible to reduce the size of the optimization space of this particular algorithm, making the tuning process faster.

Measuring and quantifying how much portable a certain optimal configuration is for different input sizes of the same problem, or for different accelerators, is

also possible within the framework of our statistical approach to auto-tuning. As for the difficulty of tuning, optimum portability is also complementary to previous research on reducing the size of the optimization space of a particular algorithm. In fact, knowing if the optimal configuration of an algorithm can be reused, in part or as a whole, in a different context may result in better pruning of the algorithm's optimization space, and impact techniques such as historical tuning [55].

To present tuning difficulty and optimum portability in a clear way, and show how these concepts look in practice, we introduce a new portable, open-source, and totally auto-tunable many-core benchmark suite called TuneBench. Therefore, we do not only define a new methodology for the quantitative analysis of auto-tuning, but we also provide an open-source tool to enable this methodology. TuneBench uses OpenCL for code portability, and auto-tuning for performance portability, and it can thus also be used to provide meaningful architectural comparisons among different accelerators. Because of its portability, TuneBench achieves higher performance on modern many-core accelerators than comparable benchmark suites like SHOC [56].

We empirically demonstrate that with our methodology, metrics, and benchmark suite, we can distinguish between applications that are easy and hard to tune. We also show *why* applications can be difficult to tune. We show that, on average, tuning memory bound applications is relatively easy, and sampling the search space of these applications during tuning is a feasible strategy. In addition, we show that for some applications, the difficulty of tuning is more dependant on the input size. Moreover, we show that the difficulty to tune can depend on different application parameters, and we quantify this effect. Finally, we compare architectures, as well as different generations of architectures from the same vendor. Therefore we make possible to answer questions like "Do new architectural features (e.g., larger caches) actually make applications easier to tune?".

To summarize our contributions, in this chapter we: (1) present a statistical and quantitative methodology to study the impact of auto-tuning, (2) introduce the concepts of tuning difficulty and optimum portability, (3) apply this methodology to study the auto-tuning of five well known applications, (4) reason about the difficulty of tuning for different classes of algorithms, and (5) release a new open-source, tunable, and portable benchmark suite for many-core accelerators.

The research question addressed in this chapter is how difficult the auto-tuning of many-core accelerators is (see Section 1.1.5). The rest of the chapter is organized as follows. In Section 7.1 we provide a brief review of the related work on auto-tuning many-core accelerators, with a particular focus on previous research about the impact of auto-tuning, as defined in this chapter. Section 7.2 introduces our methodology and the main concepts of this chapter, tuning difficulty

and optimum portability, and Section 7.3 presents the TuneBench benchmarks suite, and the applications that are part of it. We introduce the experimental setup in Section 7.4 and analyze and discuss the obtained results. Finally, Section 7.5 summarizes our conclusions.

7.1 Related Work

Auto-tuning has traditionally been used to ease the manual optimization process that is performed by programmers, and to maximize performance portability. An example is provided by [57], where the tuning of a stencil kernel for multi-core CPUs and a many-core GPU results in clear performance improvements for all platforms.

The use of auto-tuning for performance and performance portability is advocated for in [38], [11], and [13]. What differentiates this chapter from previous work is that we do not focus on a specific algorithm, but rather analyze how well-known algorithms, used in many benchmarks and applications, can benefit from tuning, and how difficult it is to tune those algorithms. This also differentiates our work from [58]. Nonetheless, we agree with the authors of [58] on the reasons why auto-tuning is important, and that one of the main challenges facing auto-tuning is the size of the optimization space. However, we believe that the size of the optimization space is just part of the problem. An equally difficult challenge is posed by the *distribution* of the valid configurations in the performance space. In this chapter, we provide insights on this distribution for different classes of algorithms and platforms, therefore allowing tuners to take a more informed decision on how, and how much, to sample the optimization space.

We agree with the authors of [59] that auto-tuning should take different input sizes into account, because the optimal configuration of an algorithm can vary as a function of the input. Our work provides further evidence to support this claim, including a quantitative analysis. These results are also relevant for tuning approaches based on historic learning, like the one described in [55]. More interesting results on the portability of optimal configurations between different inputs and platforms can be found in [60] and [61]; we agree with the authors of these papers that optimal configurations are inherently dependent on the inputs and the platforms used to run the algorithms, and provide more empirical evidences of this fact in this chapter.

Some of the algorithms in TuneBench are inspired by the SHOC benchmark [56]; while we used SHOC as an inspiration for the algorithms, our implementations have been developed by us to make auto-tuning possible. To our knowledge, the many-core benchmark suites currently available, such as SHOC [56], NUPAR [62], Parboil [63], and Rodinia [64], do not provide an easy

way to auto-tune their kernels. We and others, such as the authors of [65], believe that auto-tuning is fundamental to fairly benchmark many-core platforms. Without the performance portability provided by auto-tuning, benchmark suites would have to manually tailor their code base to each specific platform, and this is an unpractical way to solve the problem of benchmarking many-core accelerators.

7.2 Difficulty of Auto-Tuning

Assume that we have an accelerated kernel K, with a set of tunable parameters $P = \{p_0, p_1, \ldots p_n\}$; let us also assume that I is a valid input for K. In this scenario there will be one configuration c_i of the tunable parameters in P such that, if K is executed on I using c_i as its configuration, its execution time will be lower or equal than the execution time associated with all the other c_j where $j \neq i$. Auto-tuning is the process of automatically, or with minimal human involvement, finding this particular c_i that is called the *optimal configuration*. One of the main challenges of auto-tuning, and a whole research area on its own, is to find this optimal configuration, or a configuration that is close enough to it, in the least amount of time. This is usually done by reducing the size of the optimization space C, i.e. the set of all possible valid configurations of K's parameters, by applying heuristics that are specific to a kernel or to a particular input.

The reason why exhaustive auto-tuning is a time consuming process is that the performance of K needs to be evaluated for all the configurations in C, and the number of configurations depends on the size of P and the possible range of values of each p_i. Although tuning is a time consuming process, it may provide significant performance gain because the performance of K associated with its optimal configuration c_i can be much higher than the performance associated with any other configuration. So, investing time tuning a kernel is worthwhile only if the performance gain of said kernel is more important than the time spent to find the optimal configuration. Unfortunately, it is not always possible to know this in advance. However, if we knew, it would be possible to decide if finding the optimal configuration is worth the time it takes, or if sampling the optimization space to reduce tuning time would bring us a good enough configuration. Knowing how difficult tuning K is may help in taking these decisions.

In this chapter, we empirically quantify how difficult auto-tuning a kernel K is by measuring the distribution of C over performance. We say that a kernel K is *difficult* to tune if the performance associated with its optimal configuration c_i lies far apart from the performance associated with the other valid configurations in C. This definition of *tuning difficulty* is purely quantitative and statistical because it does only take into account the performance associated with each configuration in C, and is totally agnostic with respect to what these configu-

rations represent. This way, the methodology can be easily applied to kernels with many, possibly interacting, parameters, without any assumptions on their semantics. Furthermore, it can be used to reason about the difficulty of tuning entire classes of kernels, even if they do not share common parameters, and it can be also used to determine the influence that hardware platforms have on tuning itself.

The practical implications of a better understanding of tuning difficulty are multiple. Knowing that a certain kernel is easy to tune provides the tuner with a level of confidence that, even when sampling the optimization space to accelerate the tuning process, the chances of finding a configuration that is close to the optimum are high. Therefore, even when the trade-off between the performance gain brought by auto-tuning and the time that the tuning process takes is unknown, a better informed choice can be taken. Similarly, knowing that a kernel is difficult to tune may steer the tuner into not pruning the optimization space as much, because the performance gain brought by the optimal configuration is high, and chances of randomly selecting a good enough configuration are low.

Along with tuning difficulty, another important concept in our methodology for quantifying the difficulty of auto-tuning is *optimum portability*. We define optimum portability as the degree to which the optimal configuration c_i of a kernel K, running on a certain hardware platform and for a given input size, can be reused efficiently for the same kernel when running on a different platform or with a different input size. We say that the optimal configuration of a kernel K is *portable* if it can be reused, totally or partially, in a different context. This definition of optimum portability is also purely quantitative and statistical. Although optimum portability cannot be applied to entire classes of kernels, this concept can be used to reason about the reuse of a certain kernel configuration between different hardware platforms and input sizes.

Optimum portability also comes with multiple practical implications. Considered in isolation, the knowledge that a kernel is portable provides the tuner with a better starting point than the default, or empty, configuration, especially when tuning the kernel for a similar platform or a comparable input size. However, the concept of optimum portability is even more important when considered in combination with tuning difficulty. In fact, if a kernel is difficult to tune, but it is portable, knowledge accumulated from tuning it on other input sizes or platforms can be used to lower its overall tuning difficulty. Therefore a tuner can sample the optimization space of the least varying parameters, lowering the tuning time, while keeping the chances of finding a configuration that is close to the optimum high.

Though we believe that this methodology is applicable to all platforms, in this chapter we focus only on many-core accelerators. In the following sections, we empirically study the tuning difficulty and optimum portability of five different

kernels, representative of different classes of algorithms, on a variety of hardware platforms. By showing how we can quantify these concepts for some real world kernels, we also aim to gain insights into the tuning difficulty of entire classes of kernels.

7.3 TuneBench: an Auto-Tuning Benchmark for Many-Core Accelerators

In this section we introduce the benchmark suite we developed to test the methodology described in this chapter. Our benchmark suite, called *TuneBench*, is composed of five applications, is open source, and can be downloaded from GitHub*. We welcome external contributions, and see this work as the starting point of a community effort to provide more comprehensive auto-tuning and portable benchmarks for accelerators.

The tunable kernels are implemented in OpenCL, the rest of the suite is implemented in C++, and the analysis scripts are implemented in Python. The decision to implement the kernels in OpenCL has been taken to provide a common code base for the benchmarks, and avoid having different platform-specific implementations for each kernel. Four of the kernels in our suite have an equivalent in SHOC [56]; the fifth is based on the correlator algorithm described in [4]. The different applications in the benchmark have on purpose different arithmetic intensities (AI) [19]. Thus, some are clearly memory-bound, while others are more dependent on data-reuse. For some of the codes, application parameters that influence the AI are tunable. This way, the tipping point between being memory or compute-bound can be automatically localized and investigated for the different architectures by the auto-tuner.

7.3.1 Triad

The first application in our TuneBench benchmark suite is called *triad*. It is functionally equivalent to the version described in [56]. This application is the simplest in TuneBench; it takes two arrays, B and C, and sums their elements pair-wise, multiplying each element of array C by a scalar value x. The results are stored in a third array, A. Because there are only 2 arithmetic operations for every 3 memory operations, this application is completely memory bound. The input of this application is the size of the arrays. This application has three tunable parameters: p_0 represents the size of the vector used for vectorization, p_1 represents the number of work-items per work-group, and p_2 represents the number of elements processed per work-item. The performance metric used for

*https://github.com/isazi/TuneBench

this application is memory throughput, measured in bytes accessed per second. Although this application is very simple, it is a good benchmark for typical memory-throughput-bound applications, and can also be used for validation of our approach.

7.3.2 Reduction

The *reduction* application performs the reduction of an array of size n into a different array of size m, where every element of the second array is the sum of $\frac{n}{m}$ elements of the first one. Because this application performs only 1 arithmetic operation for every memory operation, it is also memory bound. The inputs of this application are the size of the input and output array; for simplicity, and functional equivalency with [56], the size of the output array is fixed at 64 for this chapter. There are four tunable parameters in this application: p_0 represents the number of elements processed per work-group, p_1 represents the size of the vector used for vectorization, p_2 represents the number of work-items per work-group, and p_3 represents the number of elements processed per work-item. The performance metric used for this application is memory throughput, measured in bytes accessed per second.

7.3.3 Stencil

In the *stencil* application, a two-dimensional stencil operator is applied to an input $n \times n$ matrix, thereby producing a $n \times n$ output matrix in which each element is a linear combination of nine neighboring input elements. In the worst case scenario, this application performs 18 arithmetic operations for every 10 memory operations, making it another memory bound application. However, by careful tuning it is possible to exploit the natural data-reuse of the application, thus reducing the number of memory operations necessary to compute neighboring output elements, while keeping the number of arithmetic operations the same. The input of this application is n, i.e. the size of the matrices. There are five tunable parameters in this application: p_0 represents the boolean condition of using local memory or device cache to exploit data-reuse, p_1 represents the number of work-items per work-group in the horizontal dimension, p_2 represents the number of work-items per work-group in the vertical dimension, p_3 represents the number of elements processed per work-item in the horizontal dimension, and p_4 represents the number of elements processed per work-item in the vertical dimension. The performance metric used for this application is arithmetic throughput, measured in arithmetic operations per second.

7.3.4 MD

The fourth application is called *MD*, and it is a n-body simulation from the field of molecular dynamics. It computes the acceleration of every atom in a closed space using the potential field generated by all other atoms in the same space; in our implementation there is no cutoff distance between atoms as in [56]. This application performs 29 arithmetic operations for every 8 memory operations, and although the ratio between these operations still makes it a memory-bound application on some platforms, it is less memory-bound than the previous applications. Moreover, also for this application it is possible to exploit data-reuse and move the bar closer to the compute-bound realm. The input of this application is the number of atoms in the simulated space. There are three tunable parameters in this application: p_0 represents the number of work-items per work-group, p_1 represents the number of elements processed per work-item, and p_2 represents the unroll factor for the innermost loop. The performance metric used for this application is arithmetic throughput, measured in arithmetic operations per second.

7.3.5 Correlator

The fifth and last application is called *correlator*. This application, from the domain of radio astronomy, computes the cross correlation between samples recorded by different radio telescope receivers; all the distinct pairs of receivers are correlated with each other. An in-depth description of the algorithm for this benchmark can be found in [4]. This application performs 32 arithmetic operations for every 8 memory operations, moving closer to being compute bound; furthermore, also for this application it is possible to exploit data-reuse and increase the ratio between arithmetic and memory operations. The inputs of this application are the number of receivers, frequencies, time samples, and polarizations; for simplicity, the values of all inputs except the number of receivers are fixed for this chapter. There are five tunable parameters in this application: p_0 represents the boolean condition of using or not constant memory to store the correlation table, p_1 represents the number of receivers to process in the horizontal dimension, p_2 represents the number of receivers to process in the vertical dimension, p_3 represents the number of work-items per work-group, and p_4 represents the unroll factor for the innermost loop. The performance metric used for this application is arithmetic throughput, measured in arithmetic operations per second.

Platform	Cores	GFLOP/s	GB/s
AMD FirePro W9100	2816	5237	320
AMD R9 Fury X	4096	8601	512
Intel Xeon Phi 31S1P	112	2006	320
NVIDIA GTX Titan	2688	4499	288
NVIDIA GTX Titan X	3072	6144	336
NVIDIA GTX 1080	2560	8228	320

Table 7.1: Characteristics of experimental platforms.

7.4 Experimental Evaluation

In this section we describe the two experiments that are the main corpus of this work, and analyze their results. But first, to address the issue of reproducibility of these same results, we introduce the platforms and the environment used to run the experiments. Table 7.1 contains the main characteristics of the six many-core accelerators used in this chapter; these characteristics are the number of cores, and the theoretical peaks for both arithmetic and memory throughput.

All accelerators are installed in computing nodes of the Distributed ASCI Supercomputer 5 (DAS-5) [66]. DAS-5 runs CentOS 7 Linux, and uses version 3.10 of the Linux kernel; the C++ compiler used is version 4.8.3 of the GCC. The AMD W9100 and Fury X use driver versions 1912.5 and 1800.11, respectively; the Intel Xeon Phi uses driver version 1.2; the NVIDIA Titan and Titan X use driver version 352.79, and the GTX 1080 version 367.35.

Due to space limits, in Section 7.4.1 we will only show the *Tuning Difficulty* results for three out of the six platforms: the AMD Fury X, the Intel Xeon Phi, and the NVIDIA GTX 1080. These platforms are, for each vendor, the most recent accelerators in our set, and thus our choice. The complete set of results, for all platforms, is available online[†]. We are, however, able to show the *Optimum Portability* results for all platforms in Section 7.4.2.

7.4.1 Tuning Difficulty

The goal of the first experiment is to present the tuning difficulty of different applications, thus providing insights on how difficult the tuning of a particular application is, as a function of the application characteristics, the platform it

[†]http://alessio.sclocco.eu/pubs/TuneBench

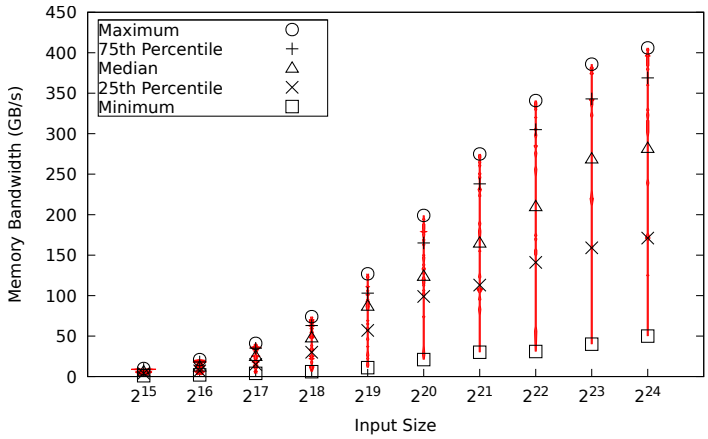

Figure 7.1: Configurations distribution, *triad*, Fury X.

runs on, and the input size. To quantify tuning difficulty we run each benchmark on all platforms, for each of the input instances, *varying all tunable parameters over all valid values*. For each run, we measure the relevant performance metric and associate its value to the configuration that generated it, thus showing the distribution of configurations over the performance space; configurations that fail to compile or run (e.g., due to memory limitations), or produce invalid results, are discarded. We perform multiple runs for each configuration, and use the average for the analysis.

All figures in this section have the same structure. The horizontal axis represents the input size, and the vertical axis represents the performance metric associated with the application (i.e., memory or arithmetic throughput). In correspondence with each point on the horizontal axis, a vertical plot shows the distribution of configurations over the performance space: the *thicker* the plot at a certain height, the more configurations are associated with that level of performance. To make the visual inspection of the figures more clear, for each distribution points are drawn in correspondence with the quartiles, so that each segment between two points contains exactly 25% of the configurations.

We begin with the results of the *triad* application, described in Section 7.3.1; the results for the Fury X are presented in Figure 7.1. While the performance range of the configurations grows as the input size increases, we notice that the distance between the maximum and the median is less than the distance between the median and the minimum, showing that the top half of the configurations are clustered more closely than the bottom half. In particular, the top 25% configurations are all close to the maximum. Figure 7.2 shows the results for the

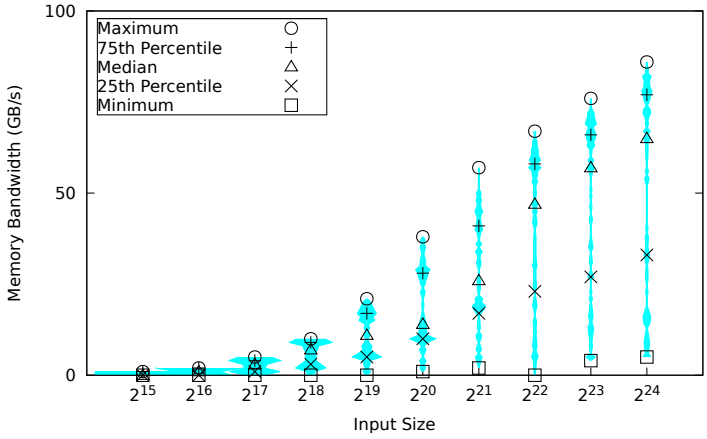

Figure 7.2: Configurations distribution, *triad*, Xeon Phi.

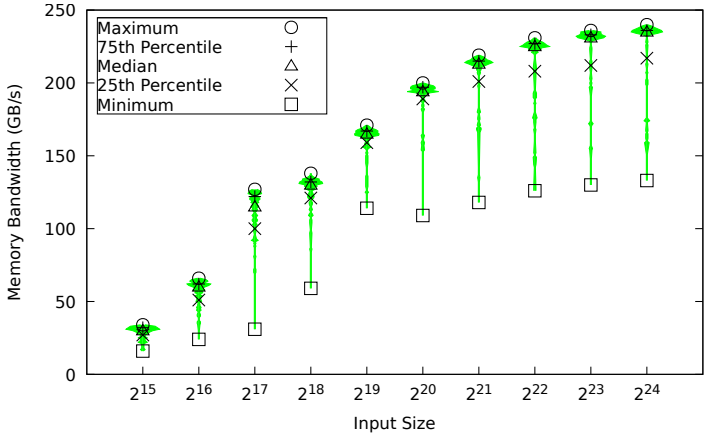

Figure 7.3: Configurations distribution, *triad*, GTX 1080.

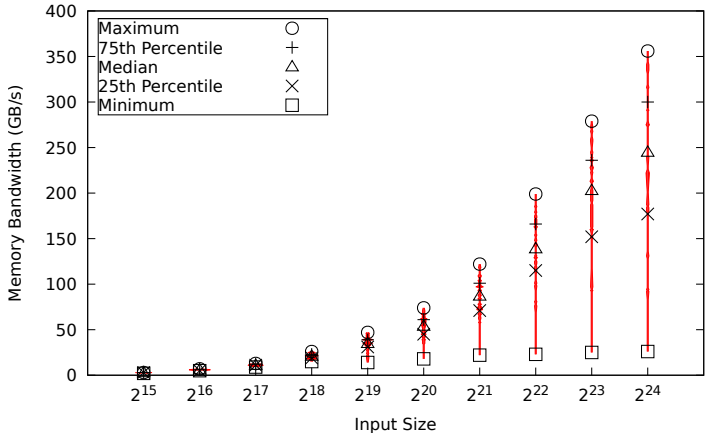

Figure 7.4: Configurations distribution, *reduction*, Fury X.

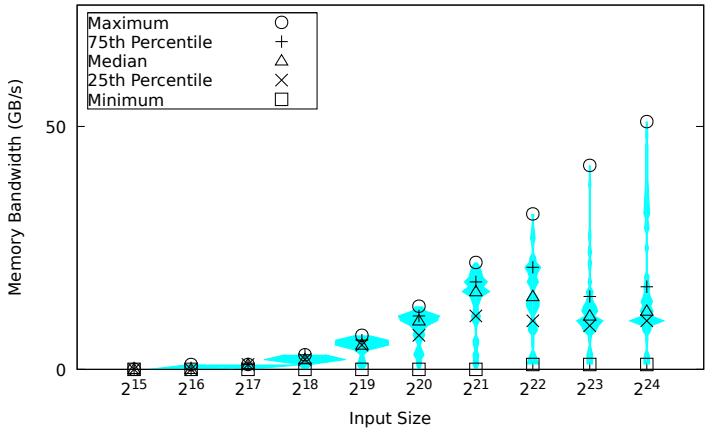

Figure 7.5: Configurations distribution, *reduction*, Xeon Phi.

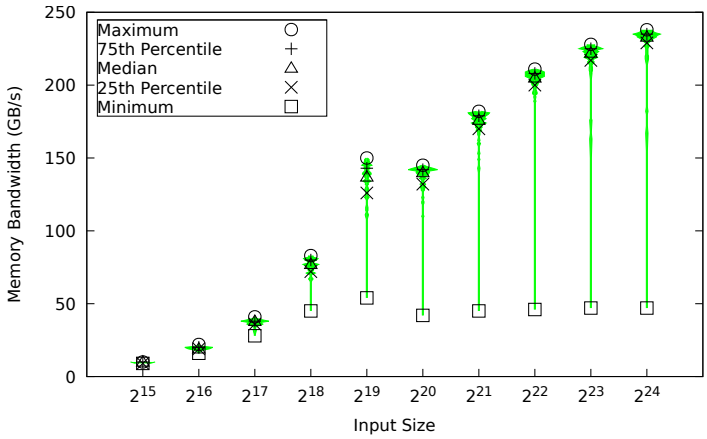

Figure 7.6: Configurations distribution, *reduction*, GTX 1080.

Xeon Phi. We observe a similar trend, although in this case the distance between maximum and median is even smaller, with the top 50% configurations close to the maximum. The GTX 1080, whose results are presented in Figure 7.3, shows an even heavier clustering around the maximum. In this case, 75% of all valid configurations are clustered at the top. Moreover, the minimums are also higher than for the previous two platforms. What we can conclude from the analysis of these distributions is that, according to the definition introduced in Section 7.2, tuning *triad* is relatively easy.

The next application is *reduction*, described in Section 7.3.2. The results of the Fury X are presented in Figure 7.4. Although similar to the previous application, these results show that the top 25% configurations are more spread, but that the overall top 75% configurations are closer. Different are the results of the Xeon Phi, presented in Figure 7.5. In this figure we see that, up to the input size of 2^{20}, the same trend is respected, but after that threshold, the maximum moves farther apart from the rest of the configurations, resulting in isolated optimums. In contrast with other two platforms, Figure 7.6 shows that for the GTX 1080 75% of the configurations achieve, in practice, performance equivalent to the maximum. For this application we can conclude that tuning is relatively easy for the GPUs; however, the Xeon Phi shows how tuning difficulty can also depend on the input size, and not just on the application or on the platform used to execute the application.

The two applications we analyzed so far, *triad* and *reduction*, are both memory bound, and their AI is not modified by tunable parameters. The distribution of configurations over the performance space, for both applications, shows that tun-

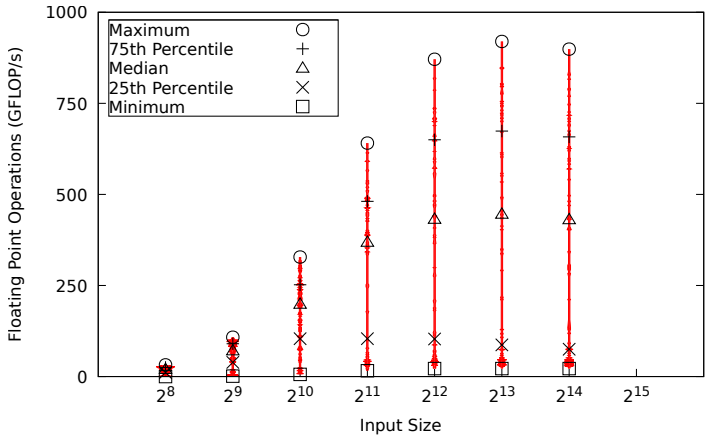

Figure 7.7: Configurations distribution, *stencil*, Fury X.

ing is relatively easy, with many configurations clustering around the maximum. As an example, we have observed that almost 75% of the GTX 1080 configurations yield performance on par with the optimum. We have also noticed, however, that there are cases in which the tuning difficulty of an application depends on the input size, as for *reduction* running on the Xeon Phi. To summarize our conclusions so far, although tuning difficulty depends on application, platform, and input size, our results suggest that tuning memory bound applications is relatively easy. Therefore, sampling the search space of these applications is a feasible strategy to reduce tuning time without excessively affecting performance.

The third application, *stencil*, was presented in Section 7.3.3. Because of possible data-reuse, it is less memory bound than the previous two. We start by introducing, with Figure 7.7, the results for the Fury X. Excluding the bottom 25% configurations, all clustered around the minimum, the remaing 75% of configurations are more evenly distributed, especially for larger inputs. Different is the case for the Xeon Phi: Figure 7.8 shows that 75% of the configurations achieve performance that is half or less the performance of the optimum. The GTX 1080 is, again, the easiest platform to tune, as can be seen from Figure 7.9. However, even if the maximum is still closer to the median than the median is to the minimum, the results are not as densely clustered near the top as they were for the previous applications. Overall, tuning this application is neither particularly easy nor hard, but definitely harder than tuning memory bound applications. We believe that the main reason for this increase in difficulty is that tuning affects the amount of data-reuse of the *stencil* application, and therefore its AI.

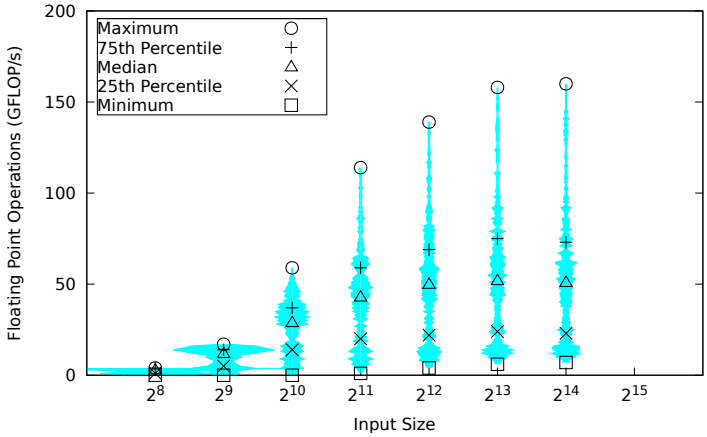

Figure 7.8: Configurations distribution, *stencil*, Xeon Phi.

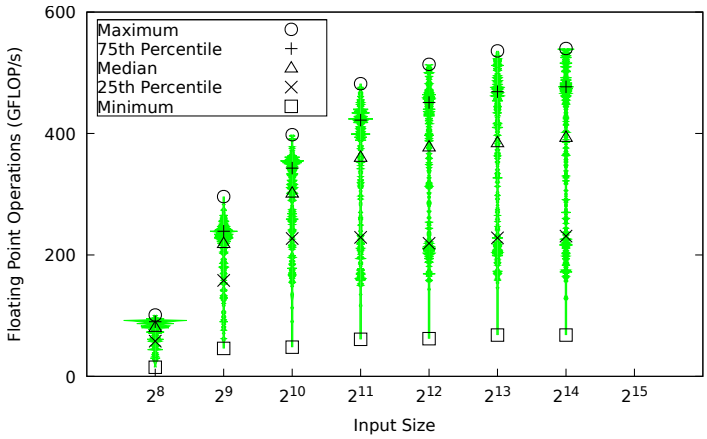

Figure 7.9: Configurations distribution, *stencil*, GTX 1080.

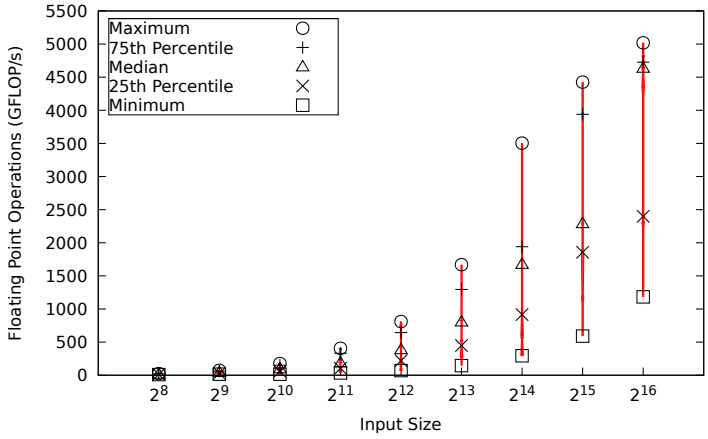

Figure 7.10: Configurations distribution, *MD*, Fury X.

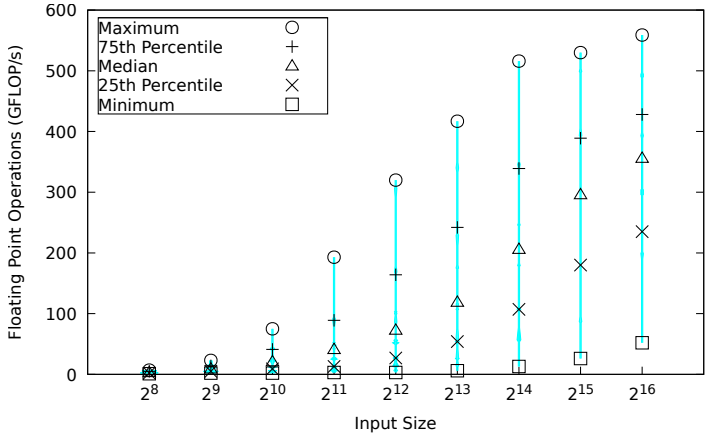

Figure 7.11: Configurations distribution, *MD*, Xeon Phi.

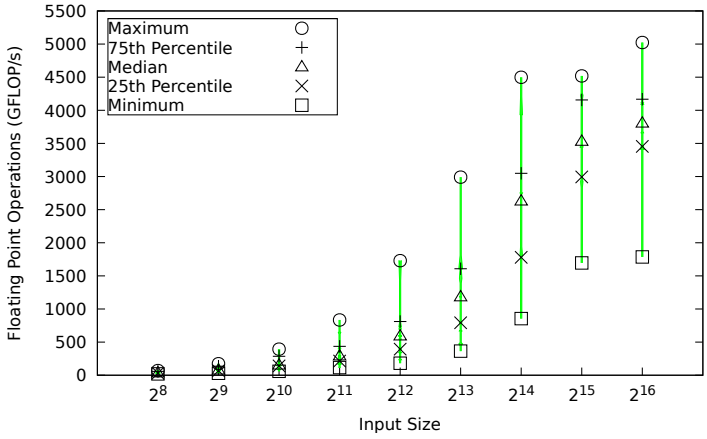

Figure 7.12: Configurations distribution, *MD*, GTX 1080.

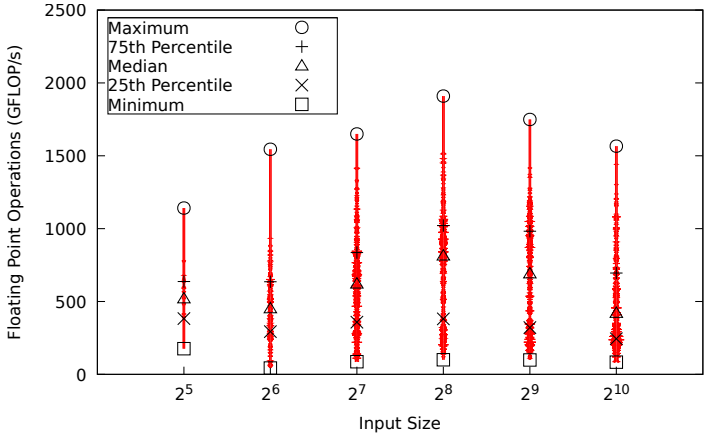

Figure 7.13: Configurations distribution, *correlator*, Fury X.

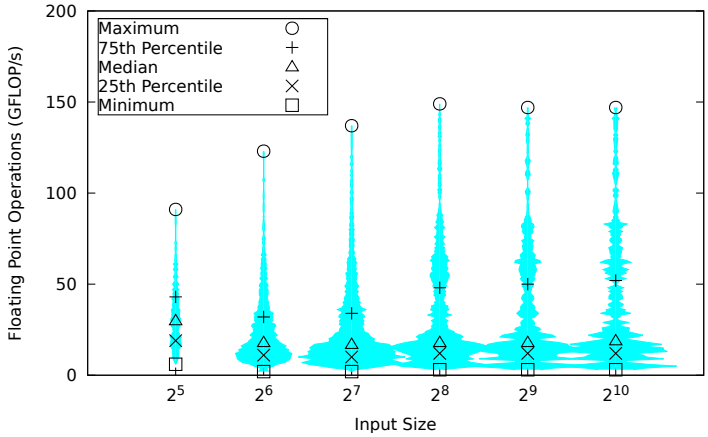

Figure 7.14: Configurations distribution, *correlator*, Xeon Phi.

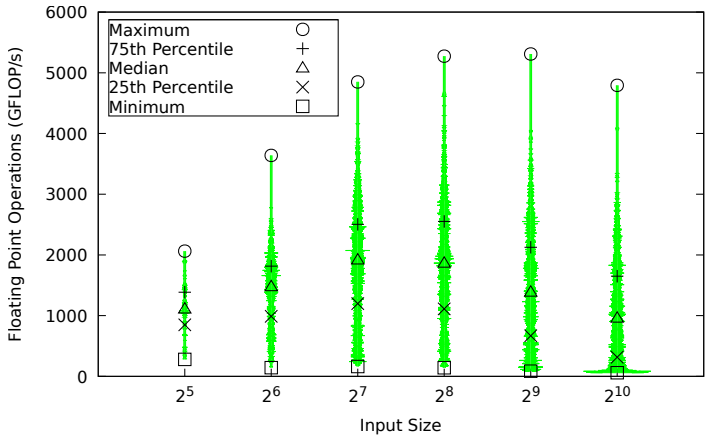

Figure 7.15: Configurations distribution, *correlator*, GTX 1080.

The fourth application in TuneBench, and the second application that is not completely memory bound, is *MD* (see Section 7.3.4). Figure 7.10 presents the results for the Fury X; here we can see that for input sizes up to 2^{15} the maximum is far apart from the median, and it can be really isolated, such as for the input size of 2^{14}. However, for the largest input size we see that 50% of the configurations cluster quite closely around the maximum, completely changing the behavior seen up to that point. In contrast, the Xeon Phi, whose results are presented in Figure 7.11, shows configurations that are spread almost evenly along the performance space, without evident clusters. Figure 7.12 shows that the GTX 1080 presents a behavior that is very similar to the Fury. In this case we see the threshold being at input size 2^{14}, with the maximum isolated up to this point, and closer to the median after. More than with previous applications, with this application we see how the difficulty of tuning can depend on the input size, and at the same time we have another example of how the GPUs and the Phi behave differently.

The last application is *correlator*, described in Section 7.3.5; like the previous two applications, *correlator* can be made less memory bound by exploiting data-reuse. Figure 7.13 shows that, for the Fury X, the maximum is isolated for all input sizes. Moreover, the top 25% configurations cover a section of the performance space that is as large as the one covered by the remaining 75% of configurations. We see the same happening for the Xeon Phi in Figure 7.14. In fact, for the Phi the performance space covered by the top 25% configurations is even larger than the space covered by the remaining configurations, and the maximum performance achieved is three times the performance associated with the 75th percentile. The GTX 1080 is no exception to this trend, as can be clearly seen in Figure 7.15. Once again, the maximum is so isolated that only a fraction of configurations are close to it. From these results, we can derive that *correlator* is the most difficult application to tune in our benchmark, and this is true for both GPUs and Xeon Phi, and for all tested input sizes. However, this is not completely unexpected: exploiting data-reuse for the *correlator* is not trivial, while performance is heavily dependent on this.

What we have seen analyzing the tuning difficulty of *stencil*, *MD*, and *correlator* is that, when tuning can be used to increase the AI of an application, the tuning process is more difficult. Of all valid configurations, only a handful have performance that are close to the optimum, and therefore sampling the optimization space to reduce tuning time is likely to have a negative impact on performance. At the same time, the advantages of finding optimal configurations are evident. Therefore, even if time consuming, an exhaustive exploration of the optimization space of this class of applications may be the best approach for performance. Another interesting result is the empirical proof that tuning difficulty can be affected by the input size of a problem, as it was the case for *MD*. Tuners

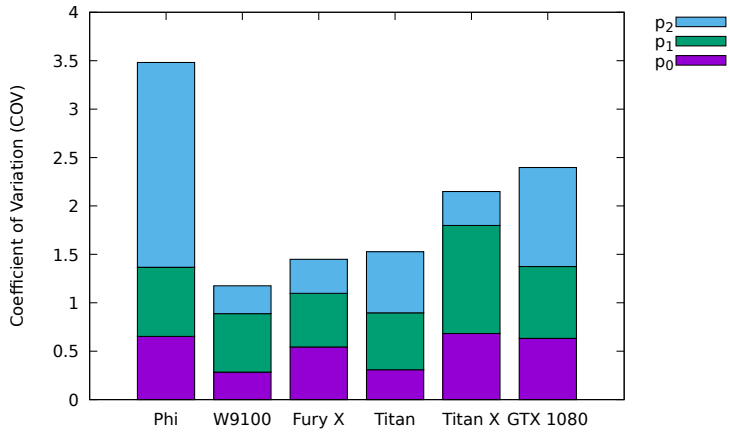

Figure 7.16: COV on different platforms, *triad*.

must pay attention to this, as a sampling strategy that produces good enough results for one input size may not be working on another one, even for the same application running on the same platform.

7.4.2 Optimum Portability

The goal of the second experiment is to present the optimum portability of different applications, and we do so by quantifying the variability of the optimal configurations, of a single application, for different platforms and input sizes. To quantify this variability of the optimal configurations we measure, and then visualize, the coefficient of variation (COV), i.e. the ratio between standard deviation and mean, of the optimal parameters tuned in the previous experiment.

The structure of all figures in this section is the same: the horizontal axis represents either platforms or input sizes, and the vertical axis represents the COV of the parameters of optimal configurations. The optimal configurations are the ones found through exhaustive tuning in the previous experiment. For each vertical bar, the COV of individual parameters is highlighted with different colors. A high COV means that the optimal value of a parameter is not stable, i.e. its value changes for different platforms or input sizes. The higher the COV, the more the value of that parameter changes; a COV of zero means that the value of said parameter does not change. What each parameter p_i represents can be found, for each application, in Section 7.3; however this knowledge is not necessary to analyze the results of this experiment, as this experiment is purely quantitative and it does not take into account the semantics of parameters.

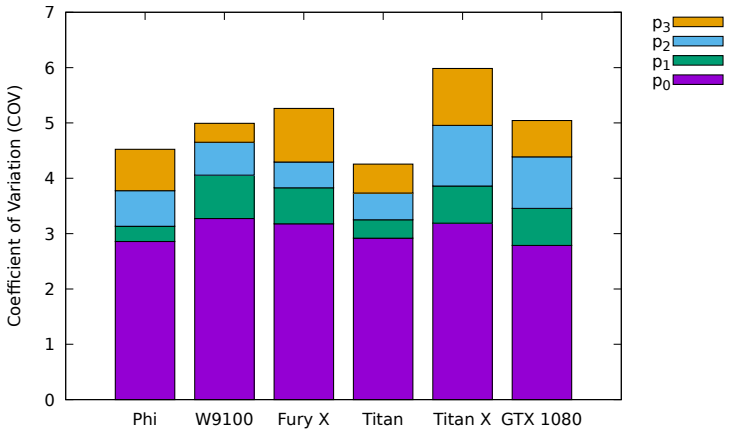

Figure 7.17: COV on different platforms, *reduction*.

Figure 7.16 shows the results for the different platforms running *triad*. The Xeon Phi has the highest COV, meaning that the optimal configurations for the different input sizes on the Phi vary sensibly in value; in particular, p_2 is the biggest contributor to the Phi's high COV, and without that contribution its COV would be on par with the other platforms. NVIDIA GPUs have higher COVs than AMD GPUs, and interestingly we see that for both vendors the COV is growing over time, with older GPUs having more stable configurations that newer ones. Even for a simple application like *triad*, new hardware opens up new valid configurations, therefore the increased variability.

Moving to *reduction*, in Figure 7.17 we see that, for all platforms, p_0 dominates the variability of the optimal configurations. There is no particular outlier this time, and all platforms have comparable COVs. The COVs would be comparable even excluding p_0, although the variability would be less in total.

In Figure 7.18 we see that there is no platform where all five *stencil*'s parameters vary. We even have a case, for the W9100, were only two out of five parameters vary, meaning that the optimal value for all other parameters is the same, on this platform, for all input sizes. In contrast, the GTX 1080 shows high variability, and it is the only platform where p_0 varies. If we exclude the GTX 1080 and the Phi, variability for this application is rather low, and it is comparable for all platforms.

In Figure 7.19 we see that the COV of *MD* is the lowest encountered so far, and that, if we exclude the GTX 1080, the other platforms have one or two parameters that do not vary at all. Again, this means that, for this application running on a specific platform, parts of the same optimal configuration can be

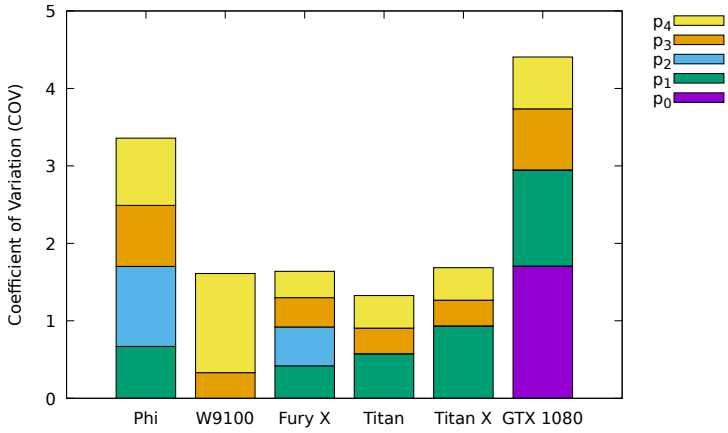

Figure 7.18: COV on different platforms, *stencil*.

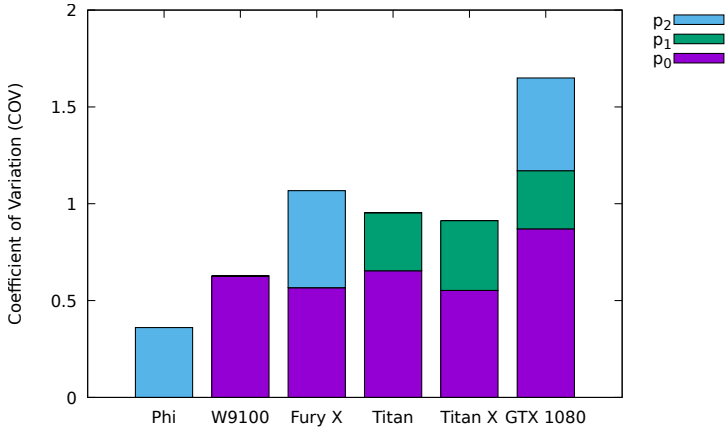

Figure 7.19: COV on different platforms, *MD*.

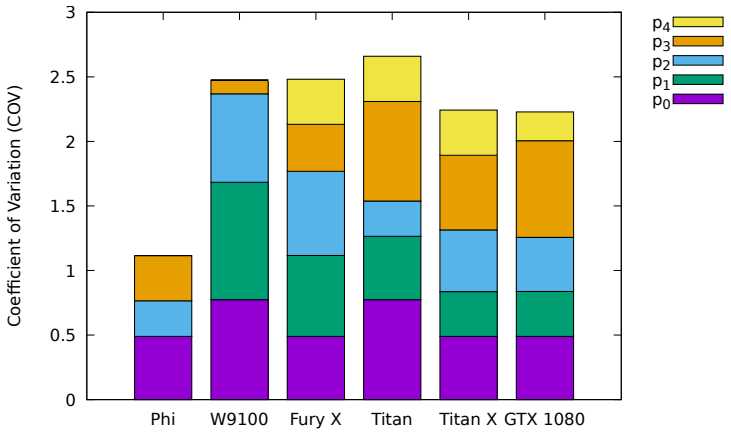

Figure 7.20: COV on different platforms, *correlator*.

used for different input sizes. The importance of this fact is more clear when combined with the results of the previous experiment. In Section 7.4.1 we noticed how input dependent the tuning difficulty of *MD* was, but now we also know that, for most platforms, the optimal configuration does not change for different input sizes. It would be therefore possible to tune input sizes that are easier to tune, and then use that knowledge, combined with the low variability of the optimal configurations, to tackle input sizes that are more difficult to tune.

The last application is *correlator*, and Figure 7.20 shows the variability for each platform. The Xeon Phi shows the lowest variability, and for this platform two out of five parameters do not vary at all for different input sizes. The variability of the GPUs is similar to each other, and twice as much the variability of the Phi. With the exception of p_4 for the W9100, all parameters do vary, although the contribution to the total variability of each parameter is different for different platforms.

To summarize the results of this experiment so far, we have seen that optimal configurations do vary, to various degree, for all tested applications, being them memory or data-reuse bound. We have also discovered that, for a given platform, it is possible at times to reuse part of an optimal configuration for different input sizes. But are optimal configurations portable, for a given input size, between different platforms? Because of space limits we cannot show all results for this experiment, and therefore limit ourself to discuss some of the results on optimum portability among platforms produced by the same vendor; the complete set of

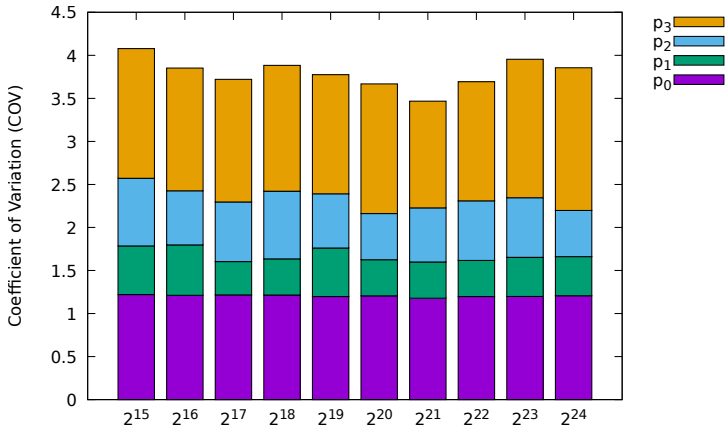

Figure 7.21: COV on NVIDIA GPUs, *reduction*.

results is available online[‡].

Figure 7.21 shows the variability of the optimal configurations for *reduction* on NVIDIA GPUs; for each input size we compare the optimum of the three NVIDIA GPUs and measure how much they vary. The figure shows how, for all input sizes, there is no part of the optimal configurations that is shared among the GPUs, confirming that this application does not offer optimum portability.

We know, from Figure 7.18, that *stencil* has a certain degree of optimum portability; Figure 7.22 contains the results for this application on AMD GPUs. What we see is that not only there is portability, for a given platform, between different input sizes, but also for a given input size between platforms from a same vendor. In fact, we can see that for some of the input sizes only one or two parameters of the optimal configurations change.

Figure 7.23 shows how variable the optimal configurations of *MD* are on NVIDIA GPUs. Again, we see that they are generally pretty variable, but there are cases in which the variability is lower. In particular, for input size 2^{15} all three GPUs share the exact same optimal configuration. Therefore, all COVs are zero, and no bar is visible.

The results for *correlator* are also interesting. Figure 7.24 shows the variability of the optimums for AMD GPUs. While we didn't see any real portability for different input sizes on a single platform, in this figure we see how portable the configurations actually are for a single input size and devices produced by a same vendor. However, the same does not apply to NVIDIA GPUs, as it can be seen

[‡]http://alessio.sclocco.eu/pubs/TuneBench

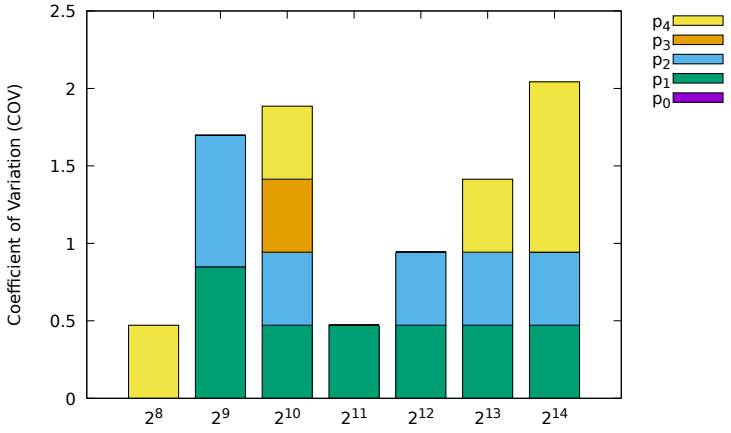

Figure 7.22: COV on AMD GPUs, *stencil*.

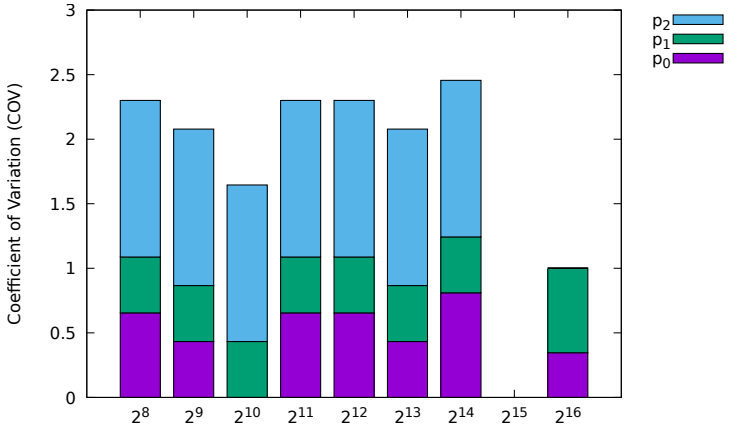

Figure 7.23: COV on NVIDIA GPUs, *MD*.

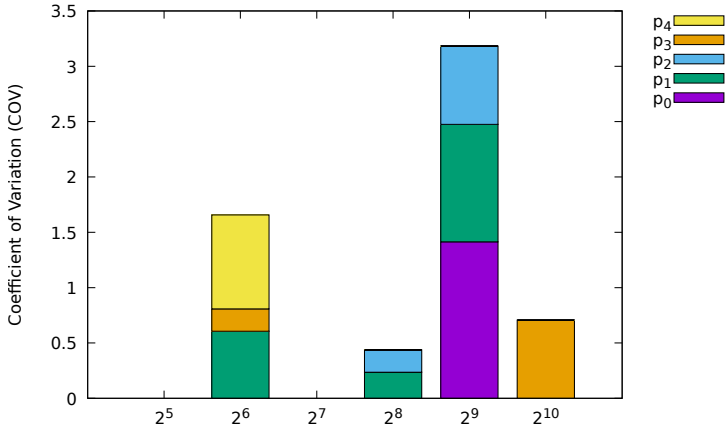

Figure 7.24: COV on AMD GPUs, *correlator*.

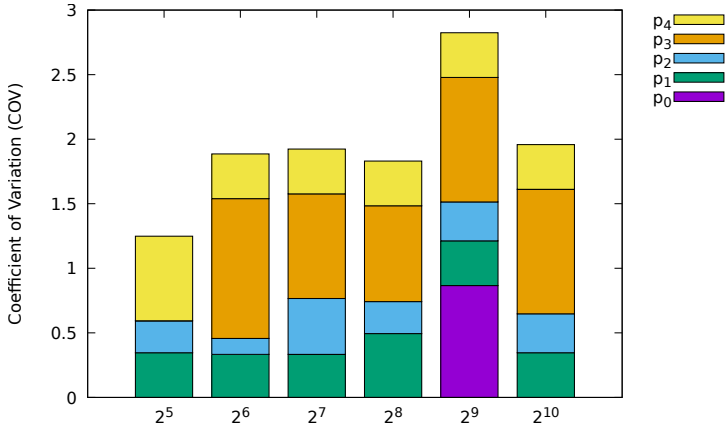

Figure 7.25: COV on NVIDIA GPUs, *correlator*.

in Figure 7.25.

Overall, achieving optimum portability remains difficult. Complex applications, running on highly parallel platforms, and working on inputs of different size, generally require different optimal configurations. However, with this experiment we showed that there are cases in which these optimal configurations can be reused, both for the same platform working on different input sizes, and for different platforms working on the same input size. Although we cannot yet link optimum portability to some characteristics of an application, like we did in the case of AI and tuning difficulty, we believe this is important for the understanding of tuning on its own. We aim to improve our understanding of optimum portability in future work.

7.5 Conclusions

In this chapter we looked at how difficult, and how important, auto-tuning is for many-core accelerators. We defined two purely quantitative concepts to improve the overall understanding of auto-tuning, *tuning difficulty* and *optimum portability*, and showed how these concepts look in practice on different parallel applications. We did not only look at different applications, and different classes of applications, but also at different hardware platforms. In fact, in this chapter we tested our methodology to statistically quantify the difficulty of tuning on six different platforms, including GPUs from AMD and NVIDIA, and the Intel Xeon Phi. The results obtained in this chapter, combined with the results obtained in Chapters 3 and 4, help us answering **RQ4** (see Section 1.1.5).

Experimental results presented in this chapter show that tuning applications that are completely memory bound is relatively easy, and definitely easier than tuning applications whose performance depend on data-reuse. In particular, NVIDIA GPUs resulted almost trivial to tune for both memory-bound applications tested, to the point that 50% to 75% of the GTX 1080's valid configurations were virtually as good as the optimum.

The same experimental results show that tuning is more difficult for applications that depend on data-reuse to achieve high performance. For these applications, we have seen that the more an increase of arithmetic intensity depends on tunable parameters, the more isolated the optimal configuration in terms of performance is. We also noticed that, for some applications, the tuning difficulty changes for different input sizes, showing how tuning is affected not only by application characteristics, but also by hardware platforms and inputs.

Improved knowledge on the difficulty of auto-tuning can be used, in practice, to steer and improve the tuning process itself. For an example, our results indicate that sampling of the optimization space when tuning memory-bound applications is a feasible strategy to reduce tuning time without compromising performance.

Although the same strategy can be applied to applications that are not completely memory bound, this may have a negative impact on performance and thus can be advised for only when a reduction in tuning time is more valuable than optimal performance.

In this chapter, we also investigated the portability of optimal configurations in two cases: between different input sizes, keeping the platform fixed, and between different platforms, keeping the input size fixed. What we observed is that there is less correlation between an application being memory or data-reuse bound and it having more or less portable optimums. However, we were able to find multiple cases in which part of the optimal configuration of a specific platform are not input size dependent, and can therefore be reused, and even cases in which the optimal configuration for a specific input size can be reused by different GPUs produced by a same vendor.

As a final contribution, we introduced TuneBench, a new open-source benchmark suite for many-core accelerators. TuneBench brings auto-tuning to many-core benchmarks, enabling more realistic, higher, and portable performance on different platforms. Furthermore, TuneBench can also be used to study auto-tuning itself, like we did in this chapter. In the future, we plan on adding more tunable applications to TuneBench, and add energy consumption to our analysis, to determine if a different optimization function does affect the tuning difficulty of an application. We also plan to test our approach on Field Programmable Gate Arrays (FPGAs), given the recent progresses made in supporting OpenCL kernels, thus measuring tuning difficulty and optimum portability on a different class of accelerators.

Chapter 8

Conclusions

At the beginning of this thesis, in Chapter 1, we defined a set of research questions, and our goal throughout the rest of this thesis has been to collect scientific evidence to answers these questions, and thus determine how to accelerate radio astronomy using auto-tuning. In Section 1.1 we presented our research questions, and explained that there are four sub-questions and how the combined answers to these questions can be used to answer the overall research question that is the subject of this thesis. In the previous chapters, we presented different many-core accelerated radio astronomy algorithms, full fledged pipelines, and the analysis of the difficulty of auto-tuning many-core accelerators; we are now ready to put together all the pieces of this scientific puzzle, and provide answers to our research questions. From Section 8.1 through Section 8.5 we provide these answers, and after we conclude the chapter with possible directions for future work.

8.1 RQ1: Can Many-Cores Be Used to Accelerate Radio Astronomy Algorithms?

In Chapter 3 we showed how the LOFAR beam former could be parallelized and tuned for multiple many-core accelerators, and we obtained a factor fifty improvement in performance and power efficiency, compared with what was the production implementation at the time of the experiments. Later, in Chapter 4, we looked at another important and widely used radio astronomy algorithm, dedispersion. What we found is that, although this algorithm is inherently memory-bound, our many-core tuned implementation can scale linearly with the size of the search space, and achieves faster than real-time performance. We also introduced some other radio astronomy algorithms in Chapter 5, 6, and 7, algorithms

such as folding, signal-to-noise ratio computation, and correlation, and also for these algorithms we did show that they can be parallelized for many-core accelerators, achieving high performance and being able to run in real-time. Overall, our selection of algorithms contains two of the most time consuming steps of the LOFAR pipeline, i.e. beam former and correlator, and the most time consuming step of the ARTS pipeline, i.e. dedispersion.

All the radio astronomy algorithms presented in these chapters benefit from being parallelized for many-core accelerators. The reasons why these algorithms thrive because of many-core parallelization are multiple. On one side, all these algorithms naturally benefit from parallelization because their input data structures can be decomposed into multiple independent dimensions, and because different output elements can be computed in parallel without explicit communication among workers. On the other side, this natural tendency to parallelism is accentuated by many-core accelerators because of the high number of cores they have, usually in the order of thousands, and from the high arithmetic throughput and memory bandwidth these platforms provide. In particular, memory bandwidth one order of magnitude higher than traditional CPUs is important, even more so for algorithms that are memory-bound. Although we did show that all the algorithms we parallelized, implemented, and tested, did benefit from many-core accelerators, among these accelerators GPUs have clearly provided the best performance results, and at the moment are the most viable platform for the acceleration of these radio astronomy algorithms.

We therefore answer this section's research question by affirming that many-cores can definitely be used to accelerate radio astronomy. This answer does not mean that we believe that many-core accelerators are the silver bullet of parallel computing, and should therefore be used to solve each and every problem. We do simply believe, and have presented results to back up our beliefs, that they are a powerful instrument that can be used to accelerate different radio astronomy algorithms, and especially those algorithms that require high performance but also adaptability to user-controlled parameters.

8.2 RQ2: What Is the Impact of Auto-Tuning on Radio Astronomy Algorithms?

In Chapter 3 we observed how auto-tuning impacts the performance of the LOFAR beam former, and in particular how important it is to find the exact right number of beams to compute in a single iteration of the innermost loop: this is important enough, in fact, to completely change the performance of the algorithm. But auto-tuning impacts more than just the performance of the LOFAR beam former, it does also affect its portability, and in this same chapter we showed how

the same code, without manual tuning or modifications, could run on multiple many-core accelerators and still achieve high performance. Similarly, in Chapter 4 we showed how auto-tuning is necessary to achieve real-time performance for another algorithm, dedispersion, and how auto-tuning can also be used to achieve performance portability between different platforms. However, for dedispersion we showed that auto-tuning can be used to achieve performance portability not only between platforms, but between different use-case scenarios, and we did show how different the optimal configurations are for different use-case scenarios. Even more important to determine the impact that auto-tuning has for radio astronomy algorithms, in Chapter 4 we showed that our auto-tuned dedispersion is much faster than a manually tuned implementation, because of the complexity of the optimization space of this algorithm, with isolated optimal configurations that are many times faster than the average configuration. To summarize, for both of the algorithms under analysis, we observed that auto-tuning had a considerable impact in terms of achieved performance, and performance portability, and that this impact makes auto-tuning an essential optimization technique. We therefore answer this section's research question by affirming that auto-tuning has a high impact on both performance and performance portability of radio astronomy algorithms.

8.3 RQ3: Are Many-Core Accelerators and Auto-Tuning Useful for Complex Radio Astronomy Pipelines?

To answer this question, in Chapter 5 we analyzed the performance, scalability, and portability of the first prototype of the ARTS radio transients pipeline. The analysis of this pipeline highlighted three important results: first that GPUs are the best candidate to accelerate the ARTS radio transients pipeline, second that the prototype scales linearly in all the search dimensions, and third that it is possible to build the ARTS transients pipeline using off-the-shelf hardware satisfying all performance, power, and budget constraints. And, by using auto-tuning, we were also able to achieve performance portability for the components, and the pipeline itself, on two different GPUs.

In Chapter 6 we performed the same analysis on a different radio astronomy pipeline, a pulsar searching pipeline; this pipeline, although sharing some of the components with the previous one, is more complex and more demanding in terms of performance, and thus interesting for our research. The main result of Chapter 6 has been to show that the real-time execution of such a complex radio astronomy pipeline is already possible today thanks to auto-tuning and many-core accelerators. As for the ARTS pipeline, we also showed linear scalability

in all search dimensions, and we used auto-tuning to adapt the pipeline to different platforms. However, for the pulsars pipeline we could perform even more experiments and show portability on GPUs and the Xeon Phi, and adaptability of the same code to three really different scenarios, including a scenario modeled on the characteristics of the first operational phase of the SKA. Comparing our results with a traditional CPU based approach, we achieve an eight time speedup in terms of performance, and a six times reduction in power consumption. All these results point us in the direction that the combined use of auto-tuning and many-core accelerators can be important not just for single algorithms, but also for more complex pipelines that are composed of multiple kernels. We therefore answer this section's research question by affirming that the combined use of auto-tuning and many-core accelerators is definitely useful for complex radio astronomy pipelines.

8.4 RQ4: How Difficult Is Auto-Tuning Of Many-Core Accelerators?

Up to this point, we concluded that auto-tuning has an impact for the performance and performance portability of radio astronomy algorithms and pipelines, and that many-cores are suitable platforms to accelerate these same algorithms and pipelines. In Chapter 4, after quantifying the impact that auto-tuning had on performance and performance portability of dedispersion, we noticed that the performance associated with the optimal configuration was many times higher than the performance associated with most of the other valid configurations, and we did notice this same fact for multiple platforms and use-case scenarios. Our conclusion, derived from this measured behavior, is that tuning dedispersion is difficult because, among hundreds of possible valid configurations of the algorithm's parameters, only a handful are associated with high performance, and the vast majority of configurations is twice as slow, or even worse, than the optimum.

We generalized this analysis to a broader range of algorithms in Chapter 7, algorithms that vary from a fairly simple memory bandwidth benchmark, to a complex radio astronomy cross correlator, and therefore quantified the impact that auto-tuning has on these algorithms on multiple platforms and for various input instances. We found differences in the difficulty of auto-tuning among different platforms, with NVIDIA GPUs easier to tune than AMD ones, and with GPUs generally easier to tune than the Xeon Phi. But we also found differences in tuning difficulty between different classes of algorithms, with algorithms that are completely memory-bound being easier to tune than algorithms that expose data-reuse, and which arithmetic intensity can be modified through tuning. We

even noticed that, for some algorithms, the difficulty of auto-tuning can be influenced by the size of the input, and that tuning the same algorithm, on the same platform, can be more or less difficult for different input sizes. Also in Chapter 7, we looked at the portability of optimal configurations among platforms and input sizes, and found out that there are cases in which it is possible to reuse all or part of an optimal configuration for different input sizes on the same platform, but that being able to reuse all or part of an optimal configuration for different platforms is far less trivial.

We therefore answer this section's research question by affirming that, in general, auto-tuning for many-core accelerators is difficult, but that to determine how difficult it is necessary to take into account multiple factors. These factors include the AI of the algorithm, the ratio between arithmetic throughput and memory bandwidth of the platform used to execute the algorithm, and the possibility of exploiting data-reuse to increase the algorithm's AI. Based on these factors, we can also conclude that tuning the two radio astronomy algorithms we presented, dedispersion and the cross correlator, on many-core accelerators is difficult, because they heavily rely on exploiting data reuse to increase arithmetic intensity, and as we have seen experimentally, their optimal configurations lie far from the rest of possible configurations in terms of performance.

8.5 How Can Auto-Tuning Accelerate Radio Astronomy?

After all the answers to the more narrow research questions, it is finally time to turn our attention to the main research question of this thesis, and determine how auto-tuning can be used to accelerate radio astronomy. First of all, to run complex pipelines in real-time, or process large amounts of data in the shortest time, radio astronomy has a need for high performance, and the most valuable way to achieve high performance today, as we explained in Section 2.3, is through heavy parallelization. So, to accelerate radio astronomy algorithms there is a need for heavy parallelization, and a suitable strategy to provide the needed acceleration, as we already demonstrated in this thesis for both standalone algorithms and complex pipelines, is through platforms such as many-cores. However, because many-core accelerators have thousands of cores and complex memory hierarchies that are not transparent to the performance-oriented user, programming these platforms presents a challenge on its own.

Our solution to overcome this challenge is to use auto-tuning, and therefore automatically find the optimal configuration to achieve high performance on a particular many-core accelerator. This same technique, auto-tuning, can also be used to adapt the same code to different platforms, input sizes, and use-case

scenarios, making it even more useful than a mere technique to improve performance. Answering the previous research questions, we already concluded that auto-tuning has a high impact on the performance and performance portability of radio astronomy algorithms and pipelines running on many-core accelerators. This impact is what makes auto-tuning particularly important for radio astronomy, and makes us affirm that auto-tuning should always be used to help accelerate radio astronomy.

To reiterate our takeaway message, and thus conclude this thesis, we believe that auto-tuning is a fundamental technique to achieve high performance on many-core accelerators, and even more so for complex radio astronomy algorithms and pipelines whose performance are limited by the amount of data-reuse that it is possible to exploit. The performance improvements gained by the correct use of auto-tuning represent, in a field like radio astronomy, the difference between being able to process in real-time the data collected by your instrument to produce high-impact science, and not being able to do so and therefore having to reduce the quality of the data and of the resulting science. Moreover, auto-tuning also represents the difference between having code that can be easily ported and adapted to new platforms, instruments, and use-case scenarios, without heavy losses in performance, and having to redevelop and manually tune your software for every possible scenario. This is why we believe that auto-tuning and many-cores should be used to accelerate radio astronomy.

8.6 Future Work

Although we just concluded that auto-tuning and many-cores should be used to accelerate radio astronomy, this does not mean that research in this field is over. Hardware is still evolving, and the many-core accelerators of today may look different from the processors and accelerators on which the algorithms presented in this thesis will run in ten years from now. During the writing of this thesis we did see GPU caches evolve, the number of cores per CPU increase, vector units get larger, 3D stacked memory hit the market, and these are only some of the changes in the high-performance computing landscape we witnessed. The years ahead will present us with plenty of new challenges: from new interconnects like Intel's Omni-Path or NVIDIA's NVLink, to accelerators that share the same silicon of their host CPUs like in AMD's Accelerated Processing Units (APUs) or the Knights Landing Xeon Phi, and from Field Programmable Gate Arrays (FPGAs) used as accelerators instead of GPUs, to the convergence of memory and computation with In-Memory Processing (IMP) platforms. Studying the performance of such diverse platforms will be one of the aspects of our work in the years to come, together with making sure that the important radio astronomy pipelines we developed will stay up-to-date and keep achieving high performance.

And although adapting to different and exotic new platforms will certainly require new algorithmic ideas and parallelization schemes, we believe that auto-tuning will remain an important tool to achieve performance portability in the future.

Auto-tuning itself will also remain an important part of our future research. The study of tuning difficulty and optimum portability can be extended to new platforms, including those FPGAs that already support the execution of OpenCL code. The benchmark suite we presented in Chapter 7, TuneBench, because of its native support for tuning of all the applications that are part of it, can be used to measure the performance of new hardware platforms more easily than traditional benchmarks that do not support tuning. In fact, extending TuneBench with more applications and analysis tools is already part of our research agenda, together with adding support for kernels written in other languages like CUDA, OpenMP, and OpenACC. Extensive tuning data obtained tuning different applications, on different hardware platforms, and running on inputs of different sizes, can also be used to understand why a particular configuration of the application's parameters is better than another one in a particular context. The analysis of such data, possibly performed by applying machine learning techniques to data we already collected, could complement the statistical analysis that we presented in this thesis, and provide new insights useful for all HPC programmers.

References

[1] Ian Bird. **Computing for the Large Hadron Collider**. *Annual Review of Nuclear and Particle Science*, **61**(1):99–118, 2011.

[2] P. E. Dewdney, P. J. Hall, R. T. Schilizzi, and T. J. L. W. Lazio. **The Square Kilometre Array**. *Proceedings of the IEEE*, **97**(8):1482–1496, Aug 2009.

[3] Gregory M Harry and the LIGO Scientific Collaboration. **Advanced LIGO: the next generation of gravitational wave detectors**. *Classical and Quantum Gravity*, **27**(8):084006, 2010.

[4] Rob V. van Nieuwpoort and John W. Romein. **Correlating Radio Astronomy Signals with Many-Core Hardware**. *Springer International Journal of Parallel Programming*, **39**(1):88–114, 2011.

[5] P. Lujan and V. Halyo. **Massively parallel computing at the Large Hadron Collider up to the HL-LHC**. *Journal of Instrumentation*, **10**(09):C09003, 2015.

[6] Ben van Werkhoven, Jason Maassen, M. Kliphuis, H.A. Dijkstra, S.E. Brunnabend, M. Van Meersbergen, F.J. Seinstra, and H.E. Bal. **A distributed computing approach to improve the performance of the Parallel Ocean Program (v2. 1)**. *Geoscientific Model Development*, **7**(1):267–281, 2014.

[7] John W. Romein, P. Chris Broekema, Jan David Mol, and Rob V. van Nieuwpoort. **The LOFAR Correlator: Implementation and Performance Analysis**. In *15th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP 2010)*, pages 169–178, Bangalore, India, January 2010.

[8] J. Mol and J. Romein. **The LOFAR beam former: implementation and performance analysis**. *Euro-Par 2011 Parallel Processing*, pages 328–339, 2011.

[9] A. Sclocco, J. van Leeuwen, H. E. Bal, and R. V. van Nieuwpoort. **A real-time radio transient pipeline for ARTS**. In *2015 IEEE Global Conference on Signal and Information Processing (GlobalSIP)*, pages 468–472, Dec 2015.

[10] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman P. Amarasinghe. **Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines**. In *PLDI*, 2013.

[11] Alessio Sclocco, Ana Lucia Varbanescu, Jan David Mol, and Rob V. van Nieuwpoort. **Radio Astronomy Beam Forming on Many-Core Architectures**. *International Parallel and Distributed Processing Symposium (IPDPS)*, 2012.

[12] Alessio Sclocco. *Radio astronomy beam forming on GPUs*. Master's thesis, Vrije Universiteit Amsterdam, May 2011.

[13] Alessio Sclocco, Henri E. Bal, Jason Hessels, Joeri van Leeuwen, and Rob V. van Nieuwpoort. **Auto-Tuning Dedispersion for Many-Core Accelerators**. *International Parallel and Distributed Processing Symposium (IPDPS)*, 2014.

[14] A. Sclocco, H.E. Bal, and R.V. van Nieuwpoort. **Finding Pulsars in Real-Time**. In *e-Science (e-Science), 2015 IEEE 11th International Conference on*, pages 98–107, Aug 2015.

[15] A. Hewish, S. J. Bell, J. D. H. Pilkington, P. F. Scott, and R. A. Collins. **Observation of a Rapidly Pulsating Radio Source**. *Nature*, **217**(5130):709–713, February 1968.

[16] R.A. Hulse. **The discovery of the binary pulsar.** In *Bulletin of the American Astronomical Society*, **26**, pages 971–972, 1994.

[17] D. R. Lorimer, M. Bailes, M. A. McLaughlin, D. J. Narkevic, and F. Crawford. **A Bright Millisecond Radio Burst of Extragalactic Origin**. *Science*, **318**(5851):777–780, 2007.

[18] Samuel Webb Williams. *Auto-tuning Performance on Multicore Computers*. PhD thesis, Berkeley, CA, USA, 2008. TUNING.

[19] Samuel Williams, Andrew Waterman, and David Patterson. **Roofline: an insightful visual performance model for multicore architectures**. *Commun. ACM*, **52**:65–76, April 2009.

[20] Jianbin Fang, Henk Sips, LiLun Zhang, Chuanfu Xu, Yonggang Che, and Ana Lucia Varbanescu. **Test-driving Intel Xeon Phi**. In *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering*, ICPE '14, pages 137–148, New York, NY, USA, 2014. ACM.

[21] AG de Bruyn, RP Fender, JME Kuijpers, GK Miley, R. Ramachandran, HJA Röttgering, BW Stappers, MAM van de Weygaert, and MP van Haarlem. **Exploring the Universe with the Low Frequency Array, A Scientific Case**, September 2002.

[22] NVIDIA Corporation. **NVIDIA CUDA C Programming Guide 4.0**, May 2011.

[23] Khronos OpenCL Working Group. **The OpenCL Specification 1.1**, September 2010.

[24] J. Roy, Y. Gupta, U.L. Pen, J.B. Peterson, S. Kudale, and J. Kodilkar. **A real-time software backend for the GMRT**. *Experimental Astronomy*, **28**:25–60, 2010.

[25] B.J. Mort, F. Dulwich, S. Salvini, K.Z. Adami, and M.E. Jones. **OSKAR: Simulating digital beamforming for the SKA aperture array**. In *IEEE International Symp. on Phased Array Systems and Technology (ARRAY)*, pages 690–694, oct. 2010.

[26] C.I.C. Nilsen and I. Hafizovic. **Digital beamforming using a GPU**. In *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 609–612, 2009.

[27] Xiaohua Lian, Homayoun Nikookar, and Leo Ligthart. **Efficient Radio Transmission with Adaptive and Distributed Beamforming for Intelligent WiMAX**. *Wireless Personal Communications*, **59**:405–431, 2011.

[28] T. Helzel and M. Kniephoff. **Software beam forming for ocean radar WERA features and accuracy**. In *IEEE OCEANS 2010*, pages 1–3, 2010.

[29] D. Byrne, M. O'Halloran, M. Glavin, and E. Jones. **Contrast Enhanced Beamforming for Breast Cancer Detection**. *Progress In Electromagnetics Research*, **28**:219–234, 2011.

[30] OpenMP Architecture Review Board. **OpenMP Application Program Interface 3.1**, July 2011.

[31] S. Thakkur and T. Huff. **Internet streaming SIMD extensions**. *IEEE Computer*, **32**(12):26–34, 1999.

[32] J. Meredith, P. Roth, K. Spafford, and J. Vetter. **Performance Implications of Non-Uniform Device Topologies in Scalable Heterogeneous GPU Systems**. *IEEE Micro*, **PP**(99):1, 2011.

[33] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen-mei W. Hwu. **Optimization principles and application performance evaluation of a multithreaded GPU using CUDA**. In *13th ACM SIGPLAN Symp. on Principles and practice of parallel programming (PPoPP)*, pages 73–82, 2008.

[34] **DAS-4: The Distributed ASCI Supercomputer version 4**.

[35] B. R. Barsdell, M. Bailes, D. G. Barnes, and C. J. Fluke. **Accelerating incoherent dedispersion**. *Monthly Notices of the Royal Astronomical Society*, 2012.

[36] W. Armour, A. Karastergiou, M. Giles, C. Williams, A. Magro, K. Zagkouris, S. Roberts, S. Salvini, F. Dulwich, and B. Mort. **A GPU-based Survey for Millisecond Radio Transients Using ARTEMIS**. In *Astronomical Data Analysis Software and Systems XXI*, 2012.

[37] Yinan Li, Jack Dongarra, and Stanimire Tomov. **A Note on Auto-tuning GEMM for GPUs**. In *Computational Science ICCS 2009*. 2009.

[38] Peng Du, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory Peterson, and Jack Dongarra. **From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming**. *Parallel Computing*, **38**(8):391 – 407, 2012.

[39] Benjamin R. Barsdell, David G. Barnes, and Christopher J. Fluke. **Analysing astronomy algorithms for graphics processing units and beyond**. *Monthly Notices of the Royal Astronomical Society*, 2010.

[40] A. Magro, A. Karastergiou, S. Salvini, B. Mort, F. Dulwich, and K. Zarb Adami. **Real-time, fast radio transient searches with GPU dedispersion**. *Monthly Notices of the Royal Astronomical Society*, 2011.

[41] D.R. Lorimer and M. Kramer. *Handbook of pulsar astronomy*. 2005.

[42] Milton Abramowitz and Irene A Stegun. *Handbook of Mathematical Functions: With Formulars, Graphs, and Mathematical Tables*. 1964.

[43] M. A. W. Verheijen, T. A. Oosterloo, W. A. van Cappellen, L. Bakker, M. V. Ivashina, and J. M. van der Hulst. **Apertif, a focal plane array for the WSRT**. *AIP Conference Proceedings*, **1035**(1):265–271, 2008.

[44] Alessio Magro, Jack Hickish, and Kristian Zarb Adami. **Multibeam GPU Transient Pipeline for the Medicina BEST-2 Array**. *Journal of Astronomical Instrumentation*, 2013.

[45] P. Chris Broekema, Rob V. van Nieuwpoort, and Henri E. Bal. **ExaScale High Performance Computing in the Square Kilometer Array**. In *Proceedings of the 2012 Workshop on High-Performance Computing for Astronomy*, pages 9–16, New York, NY, USA, 2012. ACM.

[46] Tony F. Chan, Gene H. Golub, and Randall J. Leveque. **Algorithms for Computing the Sample Variance: Analysis and Recommendations**. *The American Statistician*, **37**(3):242–247, 1983.

[47] A. J. Drake, S. G. Djorgovski, A. Mahabal, E. Beshore, S. Larson, M. J. Graham, R. Williams, E. Christensen, M. Catelan, A. Boattini, A. Gibbs, R. Hill, and R. Kowalski. **First Results from the Catalina Real-Time Transient Survey**. *The Astrophysical Journal*, **696**(1):870, 2009.

[48] J.-P. Macquart, M. Bailes, N. D. R. Bhat, G. C. Bower, J. D. Bunton, S. Chatterjee, T. Colegate, J. M. Cordes, L. D'Addario, A. Deller, R. Dodson, R. Fender, K. Haines, P. Halll, C. Harris, A. Hotan, S. Jonston, D. L. Jones, M. Keith, J. Y. Koay, T. J. W. Lazio, W. Majid, T. Murphy, R. Navarro, C. Phillips, P. Quinn, R. A. Preston, B. Stansby, I. Stairs, B. Stappers, L. Staveley-Smith, S. Tingay, D. Thompson, W. van Straten, K. Wagstaff, M. Warren, R. Wayth, and L. Wen (the CRAFT Collaboration). **The Commensal Real-Time ASKAP Fast-Transients (CRAFT) Survey**. *Pub. Astr. Soc. Australia*, **27**(3):272–282, 2010.

[49] B. Knispel, B. Allen, J. M. Cordes, J. S. Deneva, D. Anderson, C. Aulbert, N. D. R. Bhat, O. Bock, S. Bogdanov, A. Brazier, F. Camilo, D. J. Champion, S. Chatterjee, F. Crawford, P. B. Demorest, H. Fehrmann, P. C. C. Freire, M. E. Gonzalez, D. Hammer, J. W. T. Hessels, F. A. Jenet, L. Kasian, V. M. Kaspi, M. Kramer, P. Lazarus, J. van Leeuwen, D. R. Lorimer, A. G. Lyne, B. Machenschalk, M. A. McLaughlin, C. Messenger, D. J. Nice, M. A. Papa, H. J. Pletsch, R. Prix,

S. M. Ransom, X. Siemens, I. H. Stairs, B. W. Stappers, K. Stovall, and A. Venkataraman. **Pulsar Discovery by Global Volunteer Computing**. *Science*, **329**(5997):1305, 2010.

[50] W. van Straten and M. Bailes. **DSPSR: Digital Signal Processing Software for Pulsar Astronomy**. *PASA - Publications of the Astronomical Society of Australia*, **28**:1–14, 2011.

[51] J. M. Cordes. **Pulsar Observations I. – Propagation Effects, Searching Distance Estimates, Scintillations and VLBI**. In S. Stanimirovic, D. Altschuler, P. Goldsmith, and C. Salter, editors, *Single-Dish Radio Astronomy: Techniques and Applications*, **278** of *Astronomical Society of the Pacific Conference Series*, pages 227–250, December 2002.

[52] Jack W. Davidson and Sanjay Jinturkar. **Memory access coalescing: a technique for eliminating redundant memory accesses**. *SIGPLAN Not.*, **29**(6):186–195, June 1994.

[53] M. de Vos, A.W. Gunst, and R. Nijboer. **The LOFAR Telescope: System Architecture and Signal Processing**. *Proceedings of the IEEE*, **97**(8):1431–1437, 2009.

[54] P.E. Dewdney, W. Turner, R. Millenaar, R. McCool, J. Lazio, and T.J. Cornwell. **SKA1 system baseline design**. Technical report, 2013.

[55] Shahzeb Siddiqui and Saber Feki. **Historic Learning Approach for Auto-tuning OpenACC Accelerated Scientific Applications**. *The Ninth International Workshop on Automatic Performance Tuning*, July 2014.

[56] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S. Meredith, Philip C. Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S. Vetter. **The Scalable Heterogeneous Computing (SHOC) Benchmark Suite**. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU-3, pages 63–74, New York, NY, USA, 2010. ACM.

[57] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. **Stencil Computation Optimization and Auto-tuning on State-of-the-art Multicore Architectures**. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 4:1–4:12, Piscataway, NJ, USA, 2008. IEEE Press.

[58] C. Nugteren and V. Codreanu. **CLTune: A Generic Auto-Tuner for OpenCL Kernels**. In *Embedded Multicore/Many-core Systems-on-Chip (MCSoC), 2015 IEEE 9th International Symposium on*, pages 195–202, Sept 2015.

[59] Calvin Montgomery, Jeffrey L. Overbey, and Xuechao Li. **Autotuning OpenACC Work Distribution via Direct Search**. In *Proceedings of the 2015 XSEDE Conference: Scientific Advancements Enabled by Enhanced Cyberinfrastructure*, XSEDE '15, pages 38:1–38:8, New York, NY, USA, 2015. ACM.

[60] Andrew Davidson and John Owens. **Toward Techniques for Auto-tuning GPU Algorithms**. In *Proceedings of the 10th International Conference on Applied*

Parallel and Scientific Computing - Volume 2, PARA'10, pages 110–119, Berlin, Heidelberg, 2012. Springer-Verlag.

[61] Martin Tillmann, Philip Pfaffe, Christopher Kaag, and Walter F. Tichy. **Online-Autotuning of Parallel SAH kD-Trees**. In *International Parallel and Distributed Processing Symposium (IPDPS)*, 2016.

[62] Yash Ukidave, Fanny Nina Paravecino, Leiming Yu, Charu Kalra, Amir Momeni, Zhongliang Chen, Nick Materise, Brett Daley, Perhaad Mistry, and David Kaeli. **NUPAR: A Benchmark Suite for Modern GPU Architectures**. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, ICPE '15, pages 253–264, New York, NY, USA, 2015. ACM.

[63] John A. Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W. Hwu. **Parboil: A revised benchmark suite for scientific and commercial throughput computing**. *Center for Reliable and High-Performance Computing*, **127**, 2012.

[64] Shuai Che, M. Boyer, Jiayuan Meng, D. Tarjan, J.W. Sheaffer, Sang-Ha Lee, and K. Skadron. **Rodinia: A benchmark suite for heterogeneous computing**. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54, 2009.

[65] Rotem Aviv and Guohui Wang. **OpenCL-Based Mobile GPGPU Benchmarking: Methods and Challenges**. In *Proceedings of the 4th International Workshop on OpenCL*, IWOCL '16, pages 3:1–3:4, New York, NY, USA, 2016. ACM.

[66] H. Bal, D. Epema, C. de Laat, R. van Nieuwpoort, J. Romein, F. Seinstra, C. Snoek, and H. Wijshoff. **A Medium-Scale Distributed System for Computer Science Research: Infrastructure for the Long Term**. *Computer*, **49**(5):54–63, May 2016.

Summary

The goal of this thesis is to show a way to improve the performance of different radio astronomy applications. To begin with, in this thesis we advocate the use of many-core accelerators, parallel processors with hundreds of computational cores, as execution platforms for widely used radio astronomy algorithms and platforms. However, we also show that just using parallel hardware is not always enough to meet strict performance requirements. Therefore, to achieve real-time performance in the radio astronomy pipelines that are the use-cases of this thesis, we have to apply another fundamental optimization technique: auto-tuning. Auto-tuning is an optimization technique used to find the optimal configuration of a set of parameters, and in the context of this thesis we use it to find the best possible configurations of our parallels algorithms, on various many-core platforms, and for different use-case scenarios. In this thesis, by combining code generation with auto-tuning, we obtain code and performance portability for our applications, a result that is very important for a discipline like radio astronomy, where the life span of the instruments collecting data is much longer than the life span of the computers used to process these data.

In Chapters 3 and 4 we begin by showing how it is possible to improve the performance of two well-known radio astronomy algorithms, beam forming and dedispersion, by means of parallelization on many-core accelerators and auto-tuning. What we see for these two algorithms is that, both in terms of performance and energy efficiency, many-core accelerators provide better results than traditional multi-core CPUs. However, we also see that complex algorithms, running on platforms with such a high degree of parallelism, are difficult to configure and fine tune. We therefore demonstrate how auto-tuning is necessary to achieve high performance and performance portability.

In Chapters 5 and 6 we continue by showing that the combination of many-core accelerators and auto-tuning is not only beneficial for isolated algorithms, but also for more complex scientific pipelines. We do this by first looking at a prototype for the real-time pipeline of ARTS, the Apertif Radio Transient System, and then at a real-time pulsar detection pipeline, and conclude once again that

using many-core accelerators and auto-tuning it is possible to achieve real-time performance, a hard constraint for these scientific pipelines.

In Chapter 7 we conclude by showing how difficult, and at the same time how important, auto-tuning parallel applications running on many-core is. We are therefore able to generalize the importance of auto-tuning outside the domain of radio astronomy, and provide a quantitative definition of auto-tuning difficulty. We also show how this difficulty varies for different classes of algorithms, and for different platforms and input sizes.

To summarize, in this thesis we present experimental evidence that accelerating radio astronomy using many-cores and auto-tuning is a feasible and high-performance solution, and that this acceleration provides benefits that are both scientific and technological.

Curriculum Vitae

8.7 Education

Vrije Universiteit Amsterdam, Amsterdam, the Netherlands

MSc, Computer Science, August 2011

- Thesis Title: Radio astronomy beam forming on GPUs
- Supervisors: Dr. Rob V. van Nieuwpoort, Dr. Ana Lucia Varbanescu
- Grade Average: 8.8/10
- Subjects: Software Engineering, Parallel Programming, Distributed Systems

Università degli Studi dell'Aquila, L'Aquila, Italy

MSc, Computer Science, October 2011

- Thesis Title: Radio astronomy beam forming on GPUs
- Supervisors: Dr. Giudo Proietti, Dr. Rob V. van Nieuwpoort, Dr. Ana Lucia Varbanescu
- Graduation Mark: 110/110 **Cum Laude**
- Grade Average: 28.3/30
- Subjects: Software Engineering, Combinatorial Optimization, Object-oriented Programming, Formal Methods

BSc, Computer Science, March 2009

- Thesis Title: An experimental analysis of approximated TSP algorithms on graphs with sharpened triangle inequality
- Supervisors: Dr. Guido Proietti, Dr. Luca Forlizzi
- Graduation Mark: 105/110

- Grade Average: 26.3/30
- Subjects: Algorithms, Programming, Combinatorial Optimizations, Networking, Databases

Liceo Classico "G. D'Annunzio", Pescara, Italy

Maturità Classica, July 2002

- Graduation Mark: 65/100
- Subjects: Latin, Ancient Greek, Italian Literature, History, Philosophy, Mathematics

8.8 Refereed Journals

- Alessio Sclocco, Joeri van Leeuwen, Henri E. Bal, Rob V. van Nieuwpoort. Real-time dedispersion for fast radio transient surveys, using auto tuning on many-core accelerators. *Astronomy and Computing*, 2016, 14, 1-7.

8.9 Refereed Conferences and Workshops

- Alessio Sclocco, Joeri van Leeuwen, Henri E. Bal, Rob V. van Nieuwpoort. A Real-Time Radio Transient Pipeline for ARTS. *3rd IEEE Global Conference on Signal & Information Processing*, December 14 – 16, 2015, Orlando (Florida), USA.

- Alessio Sclocco, Henri E. Bal, Rob V. van Nieuwpoort. Finding Pulsars in Real-Time. *11th IEEE International Conference on eScience*, 31 August – 4 September, 2015, Munich, Germany.

- Alessio Sclocco, Henri E. Bal, Jason Hessels, Joeri van Leeuwen, Rob V. van Nieuwpoort. Auto-Tuning Dedispersion for Many-Core Accelerators. *28th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, May 19–23, 2014, Phoenix (Arizona), USA.

- Alessio Sclocco, Henri E. Bal, Rob V. van Nieuwpoort. Real-Time Pulsars Pipeline Using Many-Cores. *AAS Exascale Radio Astronomy Meeting*, 30 March – 4 April, 2014, Monterey (California), USA.

- Alessio Sclocco, Rob V. van Nieuwpoort. Pulsar Searching with Many-Cores. *Facing the Multicore-Challenge III*, September 19–21, 2012, Stuttgart, Germany.

- Alessio Sclocco, Ana Lucia Varbanescu, Jan David Mol, Rob V. van Nieuwpoort. Radio Astronomy Beam Forming on Many-Core Architectures. *26th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, May 21–25, 2012, Shanghai, China.

8.10 Invited Talks

- Tuning Difficulty on Many-Core Accelerators. *International Conference on Computing Systems (CompSys)*, June 19–21, 2017, Vught, the Netherlands.
- Analyzing the Complexity of Auto-Tuning Many-Core Accelerators. *GPGPU Systems: from hardware to programming and performance*, October 28, 2016, Utrecht, the Netherlands.
- Tuning and Evaluating Radio Astronomy Kernels. *HiPEAC Computing Systems Week*, October 7–9, 2013, Tallinn, Estonia.
- Beam Forming for the LOFAR Telescope using Many-Cores. First Workshop on High Performance Computing in Astronomy (AstroHPC 2012). In conjunction with the *21st International ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC 2012)*, June 19, 2012, Delft, the Netherlands.

8.11 Research Experience

Netherlands eScience Center, Amsterdam, the Netherlands

eScience Research Engineer **February 2017 – present**

Vrije Universiteit Amsterdam, Amsterdam, the Netherlands

PhD Candidate **March 2016 – January 2017**

ASTRON, Dwingeloo, the Netherlands

Scientific Programmer **March 2015 – December 2016**

Vrije Universiteit Amsterdam, Amsterdam, the Netherlands

PhD Candidate **September 2012 – February 2015**

Vrije Universiteit Amsterdam, Amsterdam, the Netherlands

Researcher **May 2011 – August 2012**

8.12 Teaching Experience

Vrije Universiteit Amsterdam, Amsterdam, the Netherlands

Teaching Assistant **November 2012 – February 2015**

- Teaching assistant for the *Parallel Programming Practical* (PPP) course.

Teaching Assistant **July 2012**

- Teaching assistant in the lab session dedicated to GPU programming of the Parallel Programming Summer school (in collaboration with the Universiteit van Amsterdam).

Teaching Assistant **June 2011**

- Teaching assistant in the lab session dedicated to GPU programming of the MultiMoore Summer school (in collaboration with the Universiteit van Amsterdam).

ASCI, Delft, the Netherlands

Teaching Assistant **December 2012**

- Teaching assistant in the lab session dedicated to GPU programming of the A24 course of the *Advanced School for Computing and Imaging* titled "A Programmer's Guide for Modern High-Performance Computing".

Università degli Studi dell'Aquila, L'Aquila, Italy

Teaching Assistant **January 2009**

- Teaching assistant in the undergraduate course in Software Engineering.

Cooperativa GEA, Campobasso, Italy

Internet Radio Expert **March 2008**

- Teacher in an extra-curricular course about Internet Radio and Podcasting in several secondary schools of Molise.

8.13 Awards and Honors

Faculteit der Exacte Wetenschappen - VU Amsterdam

- Scholarship 2009-2010

Azienda per il Diritto agli Studi Universitari - L'Aquila

- Scholarship 2008-2009